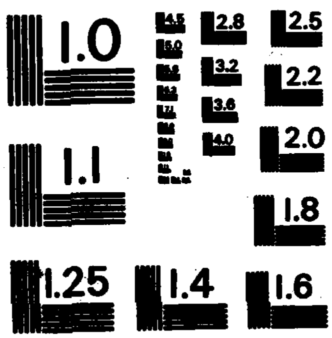
AD-A121 938 EVALUATION OF ADA AS A COMMUNICATIONS PROGRAMMING 1/4
LANGUAGE(U) SYSCON CORP SAN DIEGO CA
A L BRINTZENHOFF ET AL. 31 MAR 81 DCA100-80-C-0037

UNCLASSIFIED F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

AD A121938

REPORT DCA100-80-C-0037

EVALUATION OF ADA AS A COMMUNICATIONS PROGRAMMING LANGUAGE

ALTON L. BRINTZENHOFF
STEVEN W. CHRISTENSEN
DAVID T. MOORE
J. MARC STONEBRAKER

SYSTEMS CONSULTANTS, INC.
4015 HANCOCK ST.
SAN DIEGO, CA 92110

31 MARCH 1981

FINAL REPORT FOR PERIOD 1 AUGUST 1980 - 31 MARCH 1981

APPROVED FOR PUBLIC RELEASE, DISTRIBUTION UNLIMITED

PREPARED FOR
DEFENSE COMMUNICATIONS AGENCY
DEFENSE COMMUNICATIONS ENGINEERING CENTER
1860 WIEHLE AVE.
RESTON, VIRGINIA 22090

ATTENTION: MR. PAUL COHEN AND MS. SUSAN ZUCKERMAN

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER DCA100-80-C-0037	2. GOVT ACCESSION NO. AD-A121938	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Evaluation of Ada as a Communications Programming Language		5. TYPE OF REPORT & PERIOD COVERED Final Report 31 March 1 August 1980 - 1981
		6. PERFORMING ORG. REPORT NUMBER DCA100-80-C-0037
7. AUTHOR(s) Alton L. Brintzenhoff, Steven W. Christensen, David T. Moore, J. Marc Stonebraker		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Systems Consultants, Inc. 4015 Hancock Street San Diego, CA 92110		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Communications Engineering Center 1860 Wiehle Avenue Reston, VA 22090		12. REPORT DATE 31 March 1981
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release, Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

DCEC Contract Officer's Representatives:

Mr. Paul M. Cohen, Ms. Susan Zuckerman

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada, Communications, Communication Protocols, Concurrent programming, Segment Interface Protocol (SIP), Advanced Data Communication Control Procedure (ADCCP), CCITT High-Level Language (CHILL), Software Quality, Software Efficiency/effectiveness, trusted software

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report details the results of an evaluation of Ada as a communications programming language. This report is divided into three major sections coinciding with the efforts conducted within three separate tasks of the overall evaluation effort. The following paragraphs provide abstracts of the three sections.

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

BLOCK 20 CONTINUED

The Ada programming language promises a realistic high level alternative to the excessive cost and unreliable nature of present communication system development efforts. Using a generic communication model, the first section analyzes the ability of Ada to support communication system programming applications, especially in the area of concurrency. Previously documented criticisms as well as other problems discovered during this analysis effort are addressed. Alternatives to these problem areas are presented followed by an evaluation of the efficiency and effectiveness of the alternatives.

The CCITT High Level Language (CHILL) is being developed specifically for programming of SPC exchange applications. Ada is being developed to serve as a programming standard for embedded military computer systems. In many instances the functional requirements of these two application areas coincide and as such the second section examines the feasibility of Ada being used as a direct substitute for CHILL, both in the context of CHILL being a programming language, and in the context of CHILL being part of a programming environment containing CHILL, SDL, and MML. The report concludes that Ada is indeed a suitable replacement for CHILL in both contexts.

As part of a follow-on phase of this project, Ada will be used to implement, on a prototype basis, a communications application which consists of the AUTODIN II Segment Interface Protocol/Advanced Data Communications and Control Protocol (SIP/ADCCP) and a trusted software application which consists of the Advanced Command and Control Architectural Testbed (ACCAT) GUARD software. The third section of this report establishes the approaches to be used in designing, developing, and testing the software, evaluating the efficiency and effectiveness of Ada as used in these applications, and identifying standards and guidelines to assure overall software quality in the use of Ada.

TABLE OF CONTENTS

(This Page Intentionally Left Blank)

(This Page Intentionally Left Blank)

**AN EVALUATION OF THE
ADA PROGRAMMING LANGUAGE
FOR CONCURRENT PROGRAMMING
IN
COMMUNICATIONS SYSTEMS APPLICATIONS**

ABSTRACT

The predominant utilization of a high level language for communications systems programming applications is an attractive alternative to the current practice of machine code implementation. The Ada programming language promises a realistic high-level alternative to the excessive cost and unreliable nature of present communication system development efforts. Using a generic communication model, this report analyzes the ability of Ada to support communication system programming applications, especially in the area of concurrency. Previously documented criticisms as well as other problems discovered during this analysis effort are addressed. Alternatives to these problem areas are presented followed by an evaluation of the efficiency and effectiveness of the alternatives.

(This Page Intentionally Left Blank)

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

LIST OF ILLUSTRATIONS

(This Page Intentionally Left Blank)

EXECUTIVE SUMMARY

Using a general communication model as a basis for analysis, this report evaluates the ability of the Ada programming language to support communication system programming applications. The evaluation is directed especially toward Ada's concurrent programming features, though other advantages and disadvantages are examined as well.

Sections 2 and 3 of this report present a general communications network environment and identify the key components involved in supporting the network. In particular, specific communications functions which are implemented via software are identified and those areas which are associated with concurrency are isolated.

Section 4 examines traditional solutions to concurrent process control, i.e., interlocks, semaphores, message buffers, and monitors. Advantages and disadvantages of each mechanism are given. Ada's solution to process control, parallel tasks with entry/accept rendezvous linkage, is then described.

The material described above forms the basis for the main thrust of the analysis effort which is contained in the remainder of the report.

Section 5 examines three separate categories of potential problems associated with the use of Ada for concurrent programming applications in communication systems. The first area represents an analysis of criticisms cited within BBN Report No. 4188 /BBNE79/. The criticisms are divided into four major categories: excessive scheduler interactions, process control structure inflexibility, naming convention problems, and lack of sufficient control over the scheduling discipline. With one exception, realistic, viable alternatives are presented in answer to the aforementioned criticisms. The exception, control over the scheduling discipline, was not considered a valid criticism for reasons offered in Section 5. The second problem area concerns issues

uncovered during the analysis of Ada's ability to support the implementation of the general communication model developed in Sections 2 and 3. Again, alternatives were presented using available Ada constructs. A final problem area deals with Ada's inability to dynamically manipulate a record's structure. An alternative mechanism using unchecked conversion is offered.

In Section 6, efficiency and effectiveness criteria are defined in order to be able to evaluate the developed alternatives. Each of the alternatives is then qualitatively analyzed as to its ability to satisfactorily meet the defined criteria. In all cases, the alternatives are judged to be adequate solutions to the stated problems. In fact, the alternatives serve to point out that, as a high level programming language, Ada provides the implementor with the flexibility to construct many alternatives to presumed problem areas. A quantitative assessment of the efficiency and effectiveness of any proposed solution can only be made when a particular environment is identified and a compiler becomes available.

Conclusions are presented in Section 7. It is believed that, as a result of this preliminary analysis, the current Ada language definition can be effectively applied to communication systems programming applications.

SECTION 1
INTRODUCTION

1.1 PURPOSE

A Bolt, Beranek and Newman, Inc. (BBN) report, "The Impact of Multiprocessor Technology on High-Level Language Design," Report Number 4188 /BBNE79/, has raised several issues and identified specific difficulties which are anticipated in the use of the Ada programming language in concurrent programming applications. The purpose of this report is to address these issues and difficulties raised in the BBN report and to evaluate the efficiency, effectiveness, and problems, if any, of the Ada syntax and semantics which support concurrent programming applications.

1.2 SCOPE

The analysis of efficiency, effectiveness, and problems will be limited to those Ada areas which directly support concurrent programming applications. Other problem areas which were uncovered in this analysis will be identified and addressed. The context of the concurrency applications will be that of communication processors functioning as components of an AUTODIN II type of network.

1.3 ASSUMPTIONS

In performing this analysis, several assumptions have been made in order to provide a suitable framework for defining the problems and seeking solutions.

First, it is assumed that Ada would be applied to a state-of-the-art type communications network which is highly interconnected, employs multilayer, standardized protocols for achieving internode communications and which has demanding message volume and response-time requirements.

Second, it is assumed that a wide spectrum of computers could be used to implement the communication functions and that Ada implementations should consider the ramifications of different computer and operating system architectures.

Third, it is assumed that the Ada language should be used to the maximum extent possible in the communication software so as to achieve a high level of transportability and maintainability. Thus, applications which might normally be written in assembler code because of execution efficiency will be assumed to consist of Ada code.

Fourth, the analysis will be based on the pragmatic point of view of an implementor whose responsibility is to use the existing Ada features in the best way possible.

1.4 METHODOLOGY

The analysis of this report encompasses the dual disciplines of understanding the communication application requirements and environments as well as the Ada language syntax and semantics and the significance of various computer architectures and associated operating systems. Thus, the approach is to define the communication environment, and then identify the computer architectures, operating system features and specific communication software application functions. Next, the issues, problems, and solution alternatives are presented, evaluation criteria are defined, and the solution alternatives are evaluated. Finally, a summary of the findings is presented.

1.5 ORGANIZATION

Section 2 presents the general communications network environment and identifies the key components involved in supporting the network.

Section 3 identifies specific communications functions which are implemented via software, and isolates subsets of the software which will be affected by the concurrency issues.

Section 4 identifies the spectrum of concurrency issues generically and addresses the Ada solutions to the concurrent programming support requirements.

Section 5 identifies the BBN issues and other uncovered, related issues, defines specific problems related to each issue, and poses alternative solutions for each problem.

Section 6 establishes evaluation criteria which will be used in assessing the efficiency and effectiveness of each applicable alternative and concludes with an evaluation of the alternatives.

Section 7 summarizes the results of Section 6 and identifies any outstanding issues or problem areas.

(This Page Intentionally Left Blank)

SECTION 2
COMMUNICATION SYSTEMS BACKGROUND

The subject of communication systems software and its design is a broad one and cannot be addressed in its entirety here. However, a frame of reference is required to provide the context for discussion of the software concurrency problem with regard to current communication systems environments and software practices. To this end, this section generalizes the various aspects of communication systems and identifies what are perceived as major considerations and concepts from which issues and problems can be developed. This section addresses communication system types, software characteristics, software architectures, and future considerations.

2.1 COMMUNICATION SYSTEMS TYPES

Communication systems are employed to provide a multitude of services, which vary widely in their types of service and performance capabilities. We offer here a brief categorization of communication system types.

- Text/Source-Line Processing
- Data Acquisition/Distribution
- Process Control
- Interactive Information Processing
- Specialized Hybrid Systems
- Switching/Trunking Systems
  - Circuit Switches (AUTOVON)
  - Message (Store and Forward) Switches (AUTODIN I)
  - Packet Switches (AUTODIN II)

It should be noted that the types presented are not mutually exclusive. The more complex systems often consist of a mixture of the less complex types. Additionally, process control is used here to represent the automation of electrical, mechanical, and/or human processes. This term has a different

meaning when the topic of concurrent processing is discussed later in the document.

2.2 COMMUNICATION SYSTEMS SOFTWARE CHARACTERISTICS

We feel that communication software exhibits characteristics very similar to other "systems" software. This point of view is supported by /BBNE76/. General documentation refers to communication systems and communication applications software interchangeably. This document will refer to communication systems software rather than applications. The key point is that many applications have been written in high-level languages, while few communication systems have this distinction.

The following characteristics of communication software is evident to some degree in all the previously mentioned system types:

- Concurrent Processes
  Communication systems typically exhibit multitasking, multiprogramming qualities.
- Event Driven Operation
  Communication systems respond to events that are not directly related to the local software/hardware environment.
- Externally Performance Bounded
  Communication systems are performance bounded by factors other than local design and implementation specifications. The performance characteristics of the correspondents, the transmission facility, and the characteristics of the various protocols that are employed exhibit external performance requirements that a system must adapt to in a real-time sense.

- Transparency
  Distributed users or processes converse with each other in terms that they agree upon and understand. The intervening software and equipment is apparent only in the resulting delays encountered with data transfers.
- Service Orientation
  Communication systems provide users access to distributed processes/resources. This service has the following features:
  - Responsiveness
  - Efficiency
  - Reliability
  - Availability
  - Security
- Operating System Qualities
  Communication systems have what are classically construed as operating system qualities which are typical of "systems" software:
  - Manipulation of complex data structures
  - Maintenance of low level hardware interfaces
  - Management of local computing resources
  - High performance requirements

2.3 SOFTWARE ARCHITECTURES

Current communication software adheres to generalized, layered software architectures. This approach goes beyond the software engineering and design advantages. Such architectures transcend vendor, hardware, commercial, military, and international boundaries. The use of layered architectures provide a common approach in which dissimilar users can implement standard protocols.

This section presents a highly generalized Open System Interconnection (OSI) model, a so-called Department of Defense (DoD) model and its relationship to the general model, and a brief description of a representative implementation

currently within the Defense Communications Agency (DCA). A comprehensive model will be formulated which is an accumulation of concepts of these models and will provide the frame of reference for the remainder of the document.

2.3.1 "Reference Model for Open Systems Interconnection" (OSI) Overview

The International Standards Organization (ISO) has proposed a layered software model for general communication systems and their interconnection /OSIN79/. This model avoids references to specific protocols and embraces functional layers or protocol classes and their relationship to one another.

Although this model is aimed at the interconnection of communication system components, it also serves well to model general communication system types that have no requirement to interconnect with other system types.

2.3.1.1 Protocol Layer Description

An important aspect of the OSI architecture is that each layer of software represents a server to the adjacent superior layer. Each layer executes its protocol or functions via a set of services provided by the adjacent inferior layer. Additionally, equivalent layers across distributed components of a system form peer associations or connections. Peer associations are established, maintained, and terminated by execution of a particular protocol.

Figure 2-1 illustrates the protocol layers of the model. Specific details of this model are available in /OSIN79/ and the material is also summarized by /ZIMM80/.

2.3.1.2 Communication Systems Management Considerations

An important portion of the model is the system management structures that provide "those functions required to initiate, maintain, account for, and terminate data transfers among application processes" /ZIMM80/.

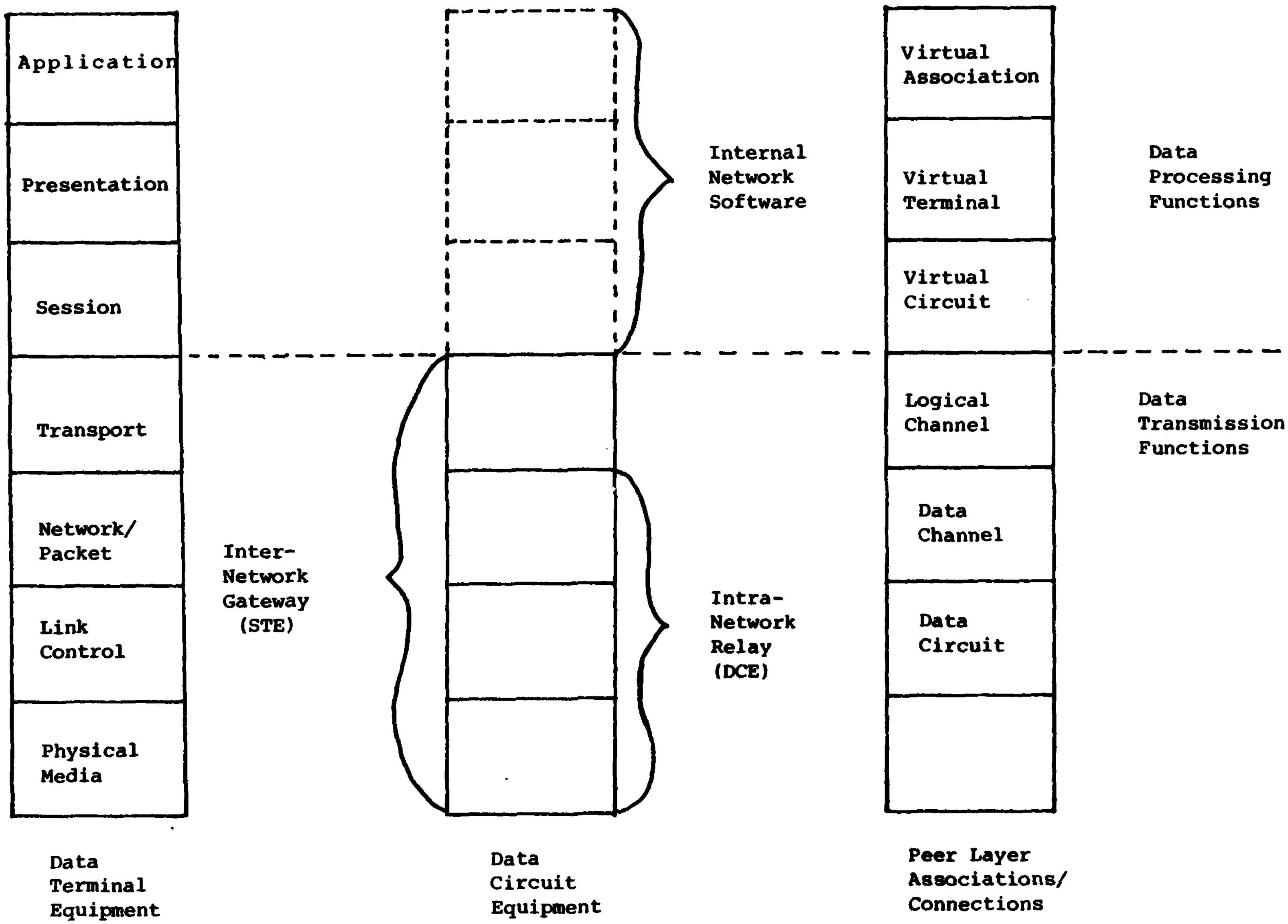


Figure 2-1
OSI Model Protocol Layer
Block Diagram

System management functions can be characterized as those that monitor, control, configure, support, and report on the system. The following list serves to illustrate the systems management functions. It should be noted that this list does not encompass all the functions required by any particular system.

- Internal and External Interface Management
- Event Management
- Resource Management
- Performance Management
- Error Management
- Recovery Structures/Procedures
- Configuration Management
- Data Management
- Test/Diagnostic Management
- Access Management

Additionally, system management functions must address two perspectives:

- Local environment or component level
- Overall system level

The component level functions address buffer acquisition, hardware and user configurations, and the operating system environment. The system level functions address connectivity to neighboring components, overall system performance characteristics, system recovery, and acquisition of system utilization statistics.

Another important aspect of the system management portion is apparent. The protocol layers address the system wide functions and processes of data transfers between distributed components. System management software provides an interface to the local operating system and hardware environment of a component of a system.

Figure 2-2 illustrates the OSI protocol layer and systems management block diagram.

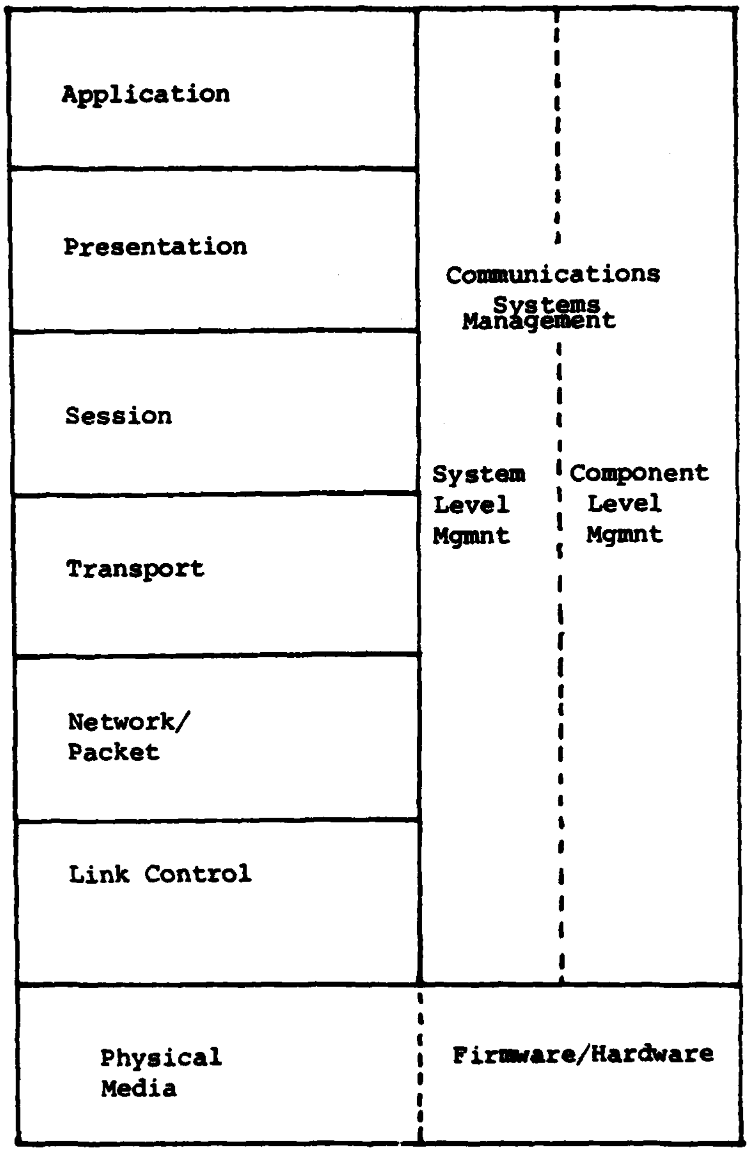


Figure 2-2
OSI Protocol Layer and Systems Management
Block Diagram

2.3.1.3 Summary

The OSI model yields a general, yet highly structured model. The model is included because the architecture that the DoD is currently adopting is based on a combination of the ARPANET structure and the OSI model. Figure 2-3 serves as a complete block description of OSI software architecture.

2.3.2 DoD Communication Architecture

This section briefly summarizes the DoD communication architecture as presented by /CLAR80/ and by /POST80/. This model is presented to provide a frame of reference for a communication system implementation of AUTODIN II. The DoD model offers a transition vehicle between the OSI model and the AUTODIN II system when discussion focuses on specific examples in the later sections.

2.3.2.1 Protocol Layer Description

At its current stage of development, the DoD model is considerably less general than the OSI model. The model does not easily provide for those systems that do not interconnect to other systems. Its development is heavily oriented in network and internetwork activities. The adoption of specific protocols such as Transmission Control Protocol (TCP) and Internetwork Protocol (IP) has resulted in the definition of sublayers rather than individual functional layers. Figure 2-4 illustrates the protocol layers of the DoD model and provides a correlation with the OSI model protocol layers.

2.3.2.2 Systems Management Considerations

The proceedings at /CLAR80/ generated no direct discussions in this area; however, numerous comments by various presenters did indicate that there is some confusion in this area. /ZIMM80/ points out that this area of communication systems is relatively undefined at present. However, the OSI

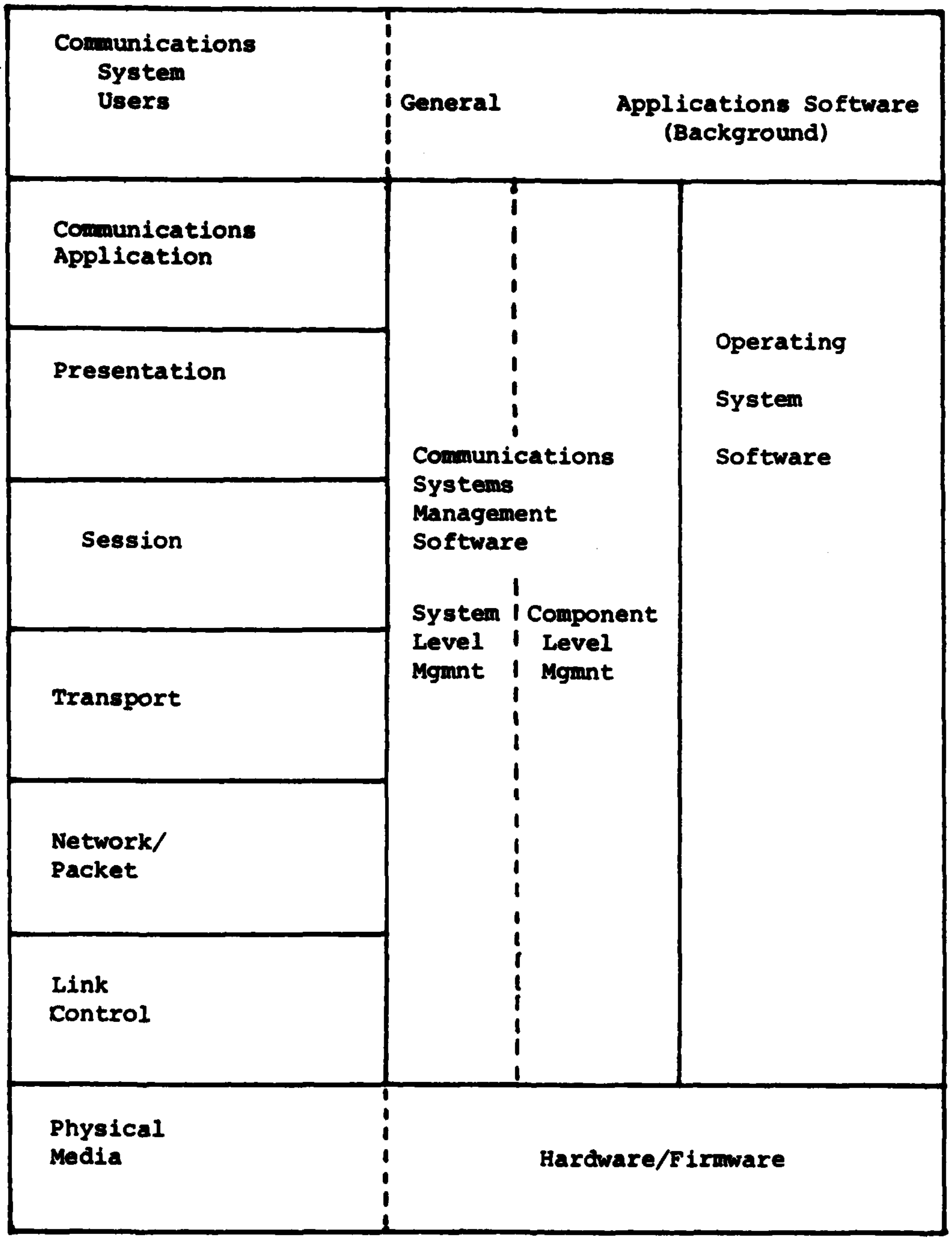


Figure 2-3
Complete OSI Model Block Diagram

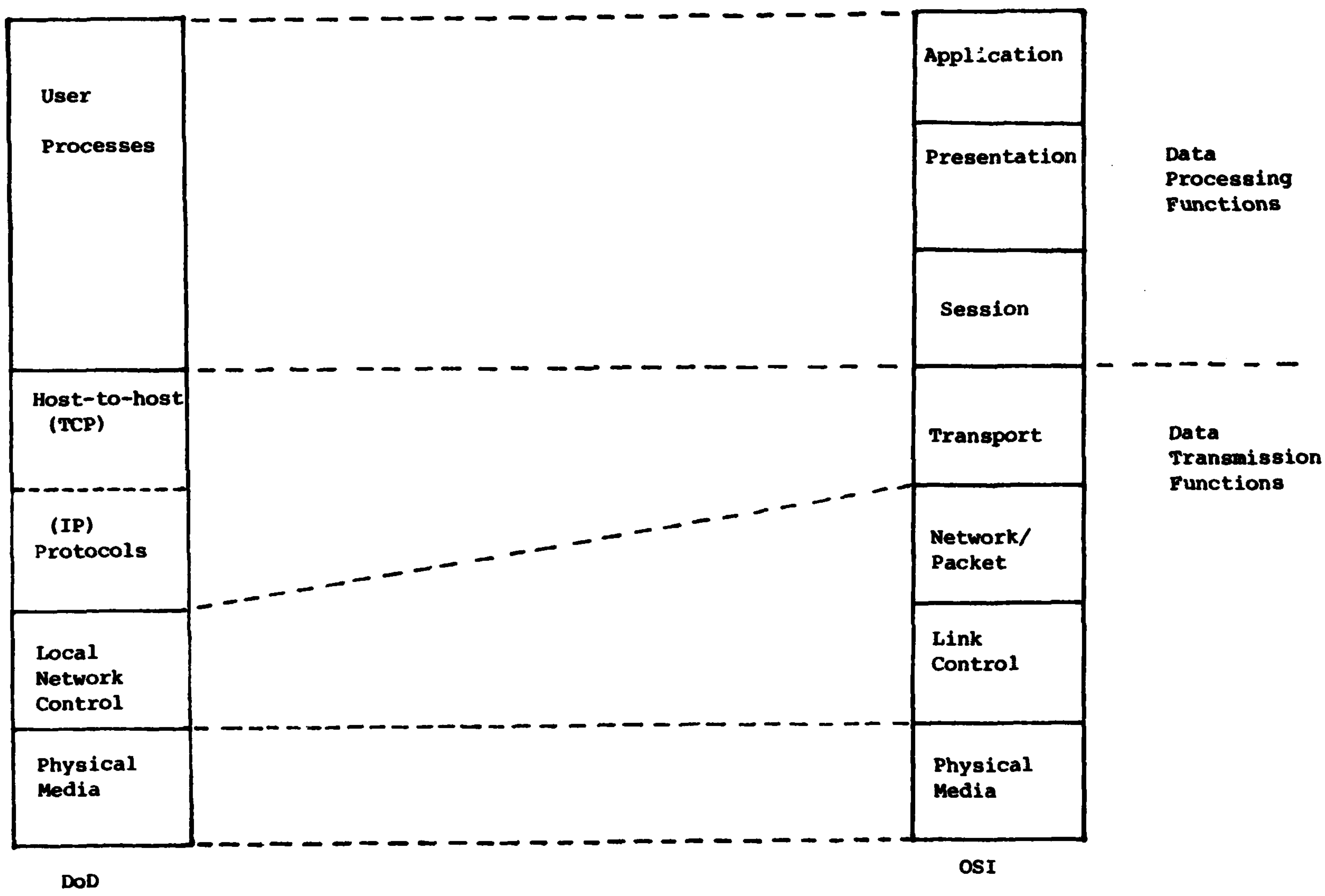


Figure 2-4
Comparison of
DoD vs. OSI Model

model has defined it generally and provided a structural block location in the model /OSIN79/.

2.3.2.3 Summary

The DoD Model can be correlated to the OSI model in a general sense, as Figure 2-4 illustrates. However, the following distinctions should be noted concerning the comparison. The DoD model is less general in nature; it deemphasizes the strong connection orientation of the OSI model, it is more prone to sublayering as opposed to the definition of precise functional software divisions; and it does not generally address systems that are not interconnected.

2.3.3 AUTODIN II System Overview

This section of the document is provided to establish a correlation between a member of the DCS community and the more general architectures. This treatment is derived from /AUTO78/.

The communication system frame of reference is narrowed to that of packet switches generally and to an AUTODIN II type of system specifically. The reasons for this are as follows:

- Packet switches are replacing other trunking types of systems.
- Packet switches utilize the entire OSI and DoD architectural models.
- Packet switches exhibit severe performance requirements.
- A portion of the AUTODIN II system, or an interface to it, will be implemented in the Ada language as a practical evaluation of Ada in the context of communication software.

Figure 2-5 serves as a general functional topology of the AUTODIN II system. It in no way implies specific geographical or network configurations.

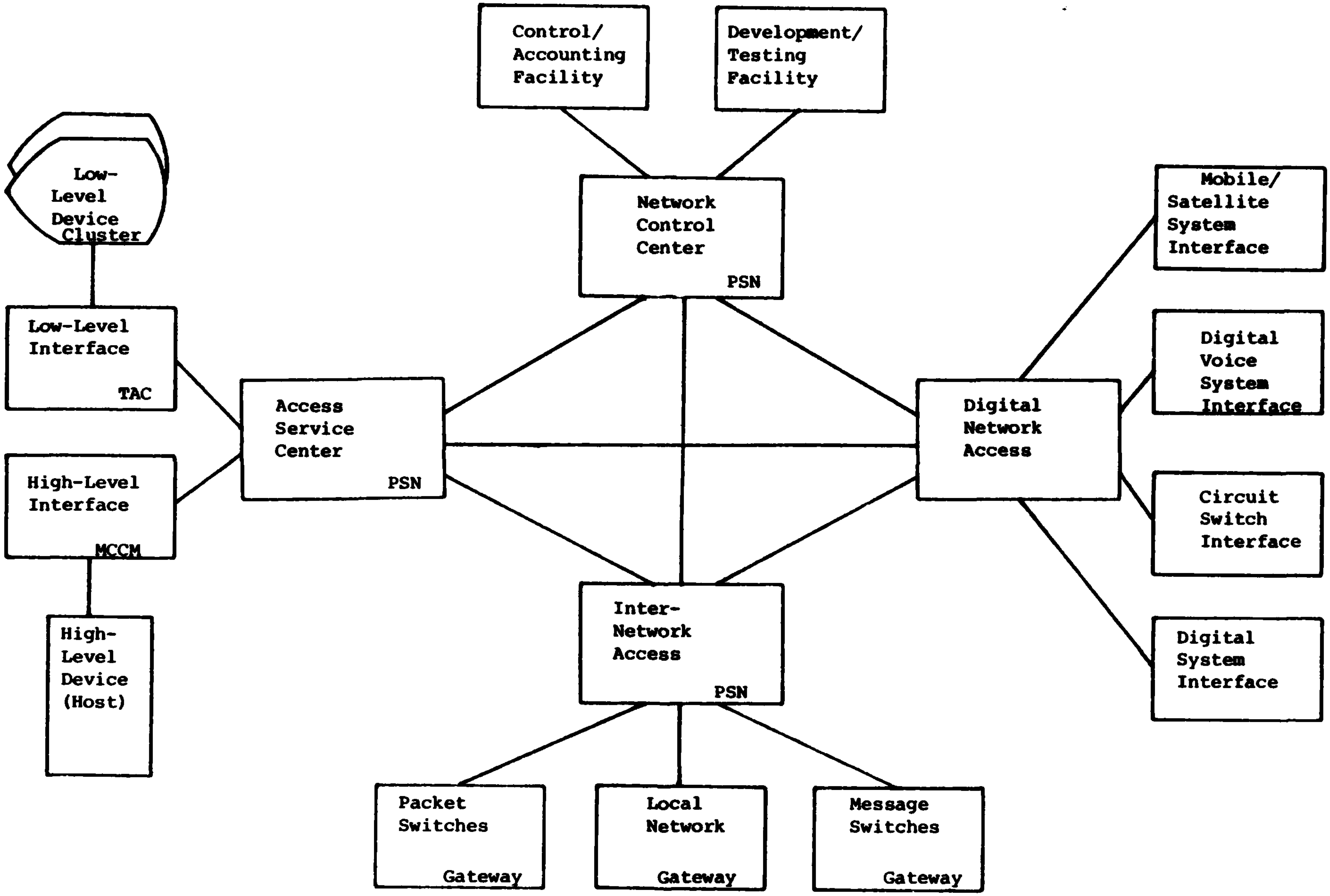


Figure 2-5
Functional Topology for
SCI System

2.3.3.1 Functional/Protocol Layer Description

Figures 2-6 and 2-7 depict the protocol layers for various components of the system. The diagrams do not represent the Control and Test facilities interface points. These facilities and their functions are not required as a portion of the protocol layers. The general protocol descriptions are as follows:

- Transmission Control Protocol
  This function generally manages a connection between correspondents. This involves data transfer, control, and synchronization at the user message level.
- Segment Interface Protocol
  This function controls data transfers between access area (user or data environment) and the network area (transmission facility environment).
- Terminal Interface/Host Interface
  This function is a set of protocols and signaling conventions that correspond to particular terminal and host classifications or sets, (i.e., RS-232, MIL-STD-188-114, IBM channel interface, etc.)
- Terminal/Host Protocol
  Two functions are provided at this level. One function establishes the terminal/host interface characteristics and the other function establishes the formats for data exchange using the established characteristics.
- Source-Destination Protocol
  This function provides addressing, routing, and control functions which direct traffic across the network.

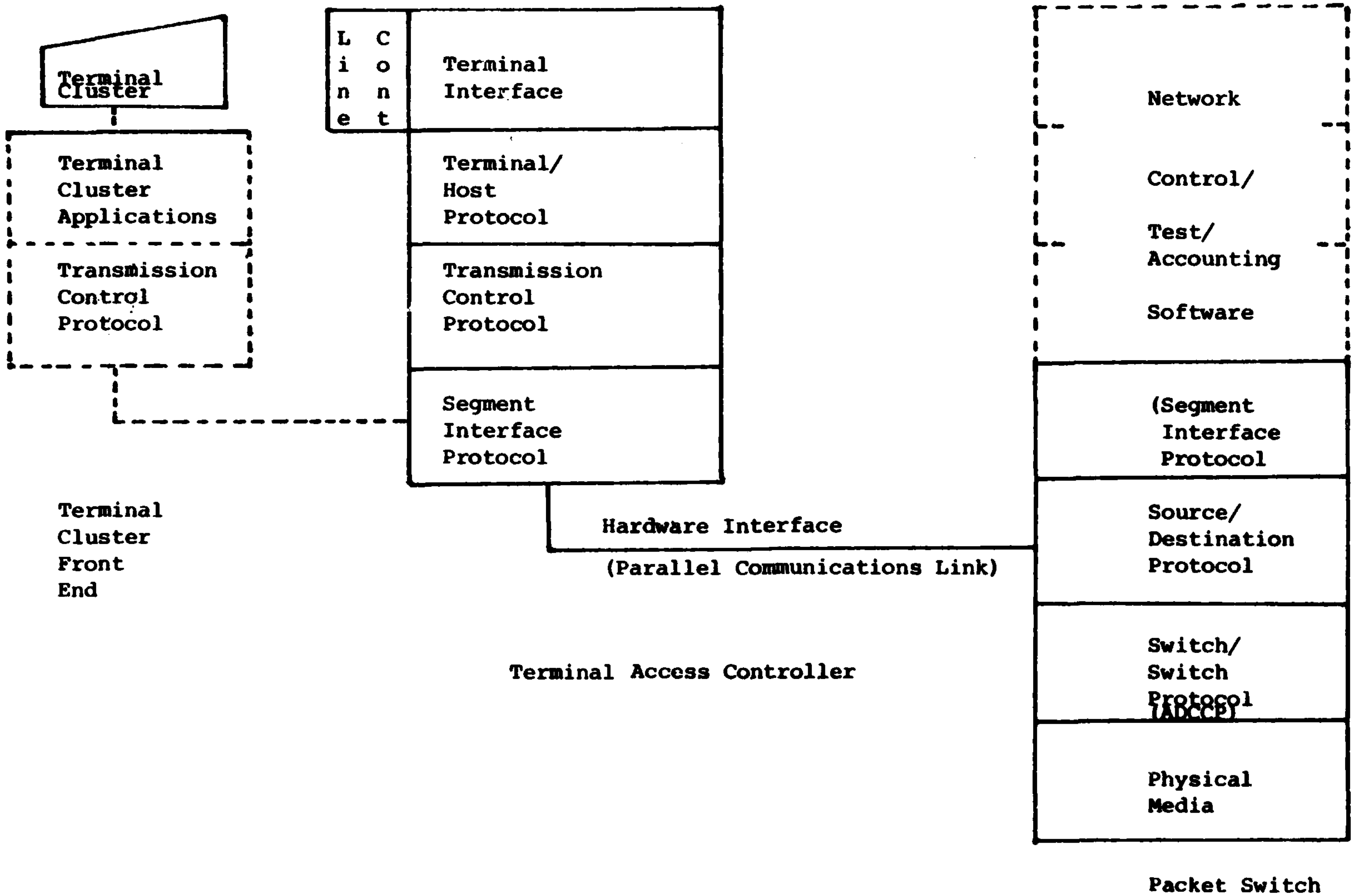


Figure 2-6
AUTODIN II Terminal Interface (TAC)
Protocol Structure

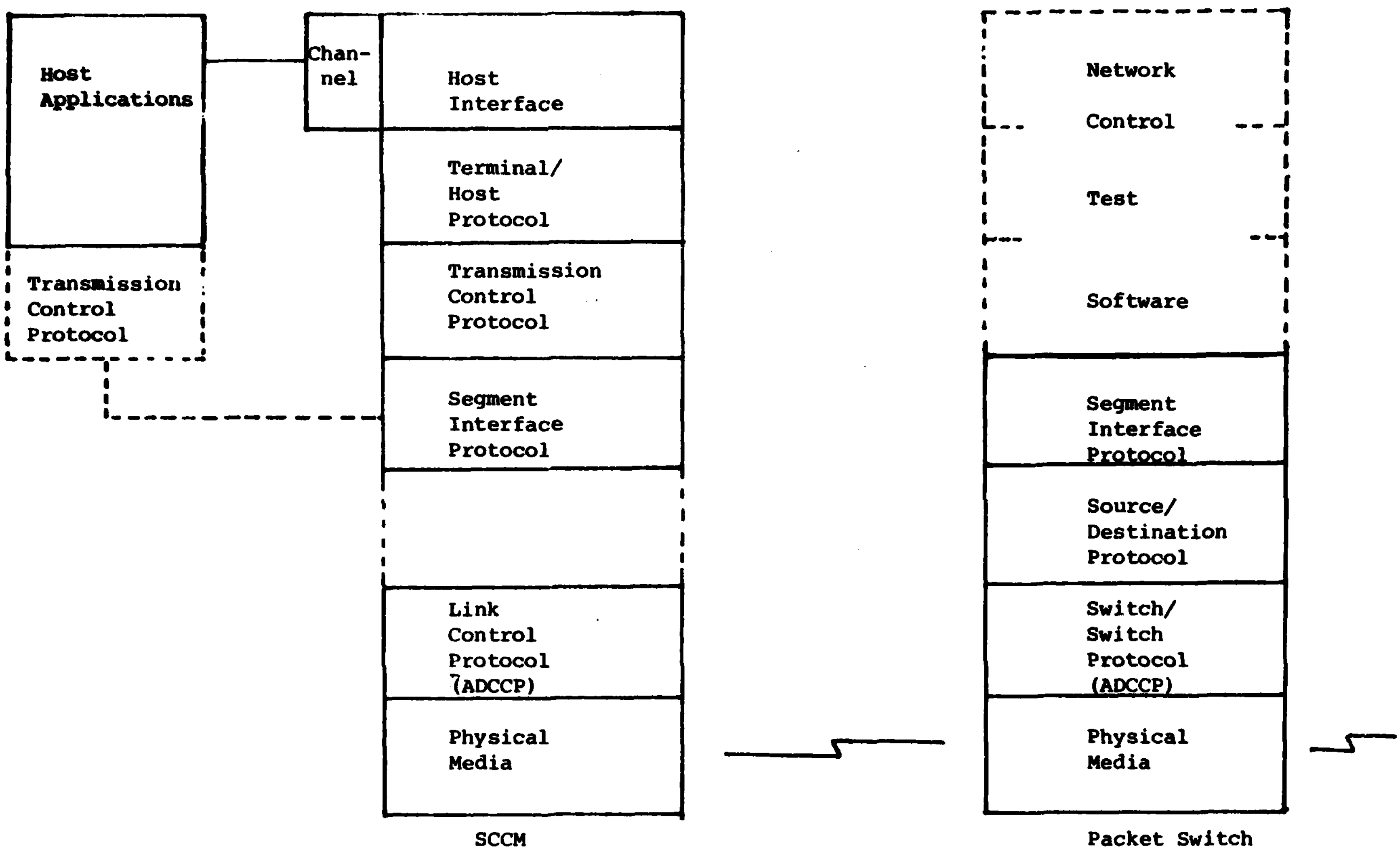


Figure 2-7

AUTODIN II Host Interface (SCCM)
Protocol Structure

- Switch-Switch Protocol
  This function provides the line control procedures necessary to establish, maintain, and release an Advanced Data Communications Control Procedure (ADCCP) protocol type of link between adjacent switches.
- Physical Media
  This layer is perceived as the electrical, mechanical, and procedural requirements of the hardware data circuit.

2.3.3.2 System Management Functions

The AUTODIN II model, like the DoD model, lacks completeness in this important area. We must again draw on the OSI model for discussion involving system management software. By using the OSI, DoD, and AUTODIN II models, it is now possible to form a composite model which describes for AUTODIN II not only the protocol layers and functions, but also the system management functions. Figure 2-8 illustrates the architectural correlation between the OSI, DoD, and AUTODIN II models.

2.4 FUTURE CONSIDERATIONS

Communication protocol standards are emerging within the framework of the architectural models. These standards are propagating upward through the architectures.

Thus, the models serve not only as a convenience from the software engineering point of view, but also are a framework from which wider interconnectivity is possible between dissimilar users.

Another important aspect of protocol standards is apparent. As standards are adopted, the software issue becomes one of implementation rather than design. The rapid increases in hardware technologies and performance along with the rapid decrease in costs makes a hardware implementation of a communication protocol a very attractive consideration. Thus,

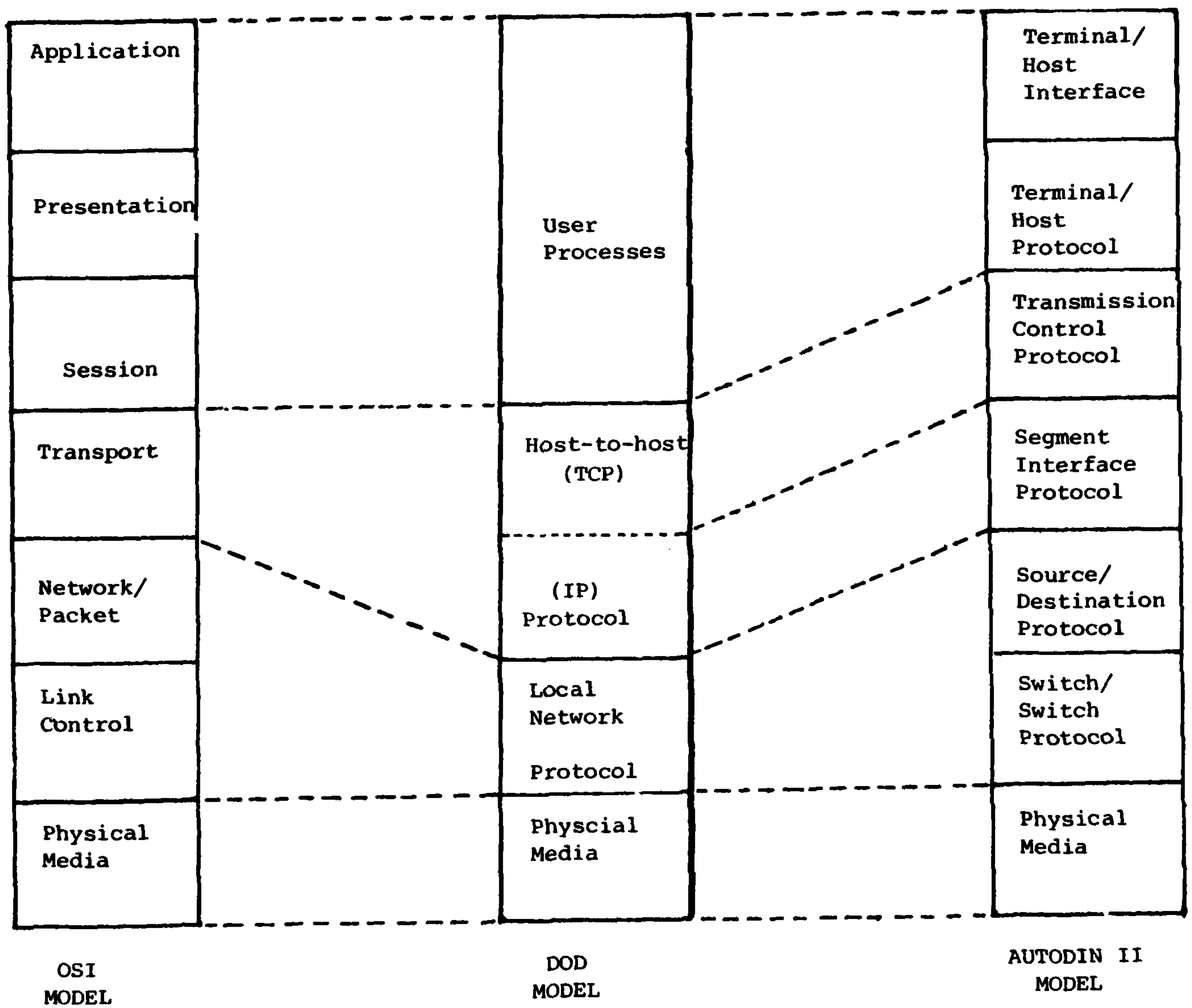


Figure 2-8
OSI/DOD/AUTODIN II MODELS
Protocol Layer Comparison

lower-level protocol layers could essentially disappear from the classical software implementation.

2.5 CONCLUSIONS

Using existing communication system models, a communication software model which is representive of current communication systems software architectures has been formed and is illustrated in Figure 2-9. This model, which we will refer to as the SCI architecture, will form the frame of reference for the remainder of the document. The SCI architecture is based on the OSI, DoD, and AUTODIN II models. It exhibits a highly modular, hierarchical structure. The identified modules possess functional orientations. The SCI architecture implies general user/server types of intermodular relationships. A high-order language implementation should generally map onto the SCI architecture.

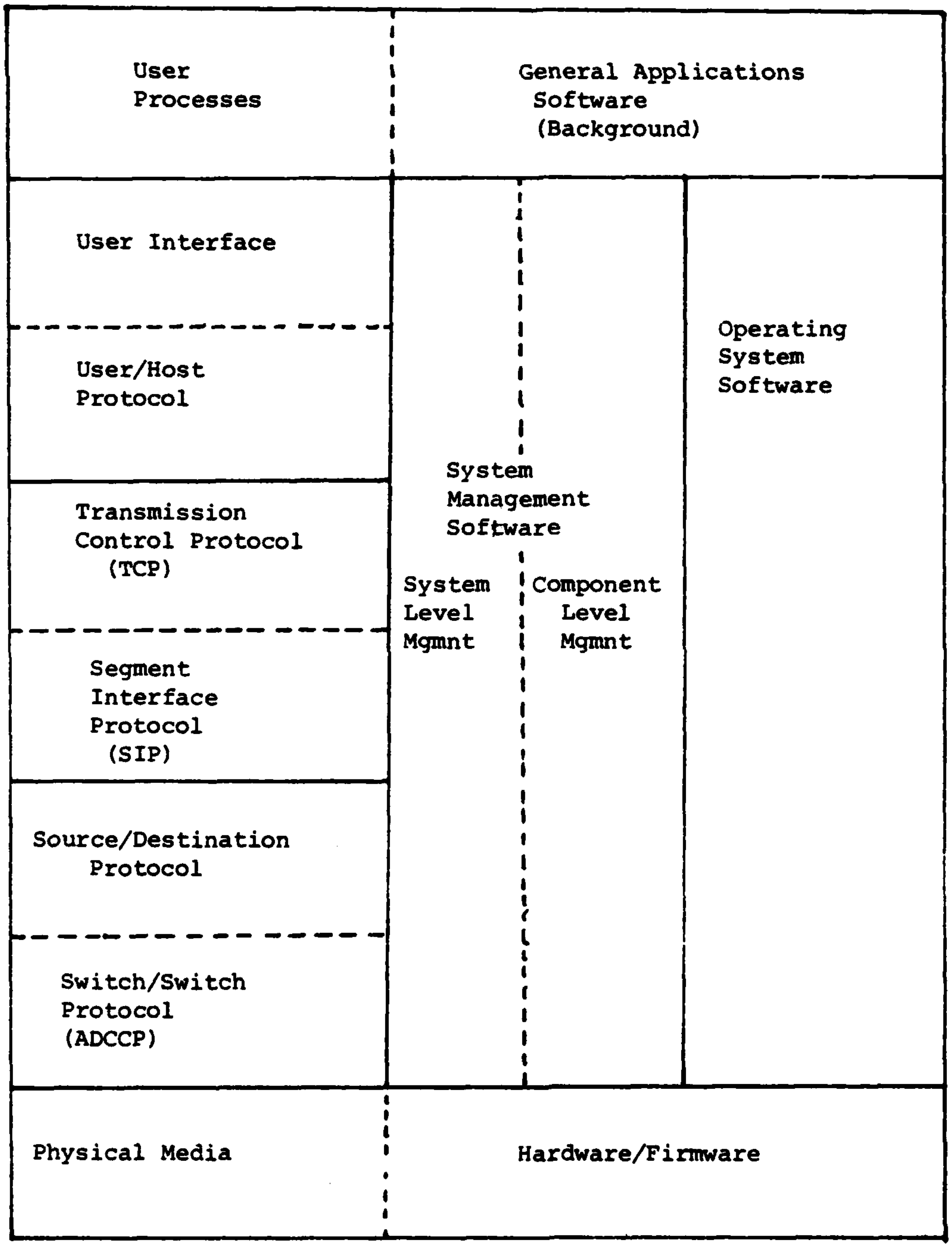


Figure 2-9

SCI Architecture Block Diagram

(This Page Intentionally Left Blank)

SECTION 3
COMMUNICATION SYSTEMS ENVIRONMENTS AND PRACTICES

This section describes the hardware/software environments and implementation practices in the development of systems based on the SCI architecture. This will be accomplished by providing an additional level of detail to the discussion of Section 2. This section will establish an environment from which to address the concurrent processing considerations. In addition, issues that are not directly related to concurrent processing that are deemed important and warrant discussion will also be presented.

The following topics are addressed: performance considerations, hardware considerations, architectural considerations, and software engineering considerations.

3.1 PERFORMANCE CONSIDERATIONS

An AUTODIN II type system exhibits severe performance requirements. In this section we will identify the performance environment in which the SCI system must operate.

Generally, a communication systems performance is a measure of its responsiveness to user stimuli and the number of users it can support. Figure 3-1 is an illustration of the relative performance requirements on packet switch nodes. As the graph indicates, the nodes operate in a narrow band around the point of optimization. The AUTODIN II system handles mixed query/response and bulk traffic. Query/response traffic requires a rapid response time. Bulk traffic, alternatively, requires high throughput capacity. The point of optimization is the point where the system is utilized efficiently while allowing temporary excursions above and below without saturating or grossly underutilizing the system.

The sequential orientation of the protocol layers addresses the response time requirements of the system. Typically, the AUTODIN II system is required to transfer a high priority query/response message across the network within three seconds.

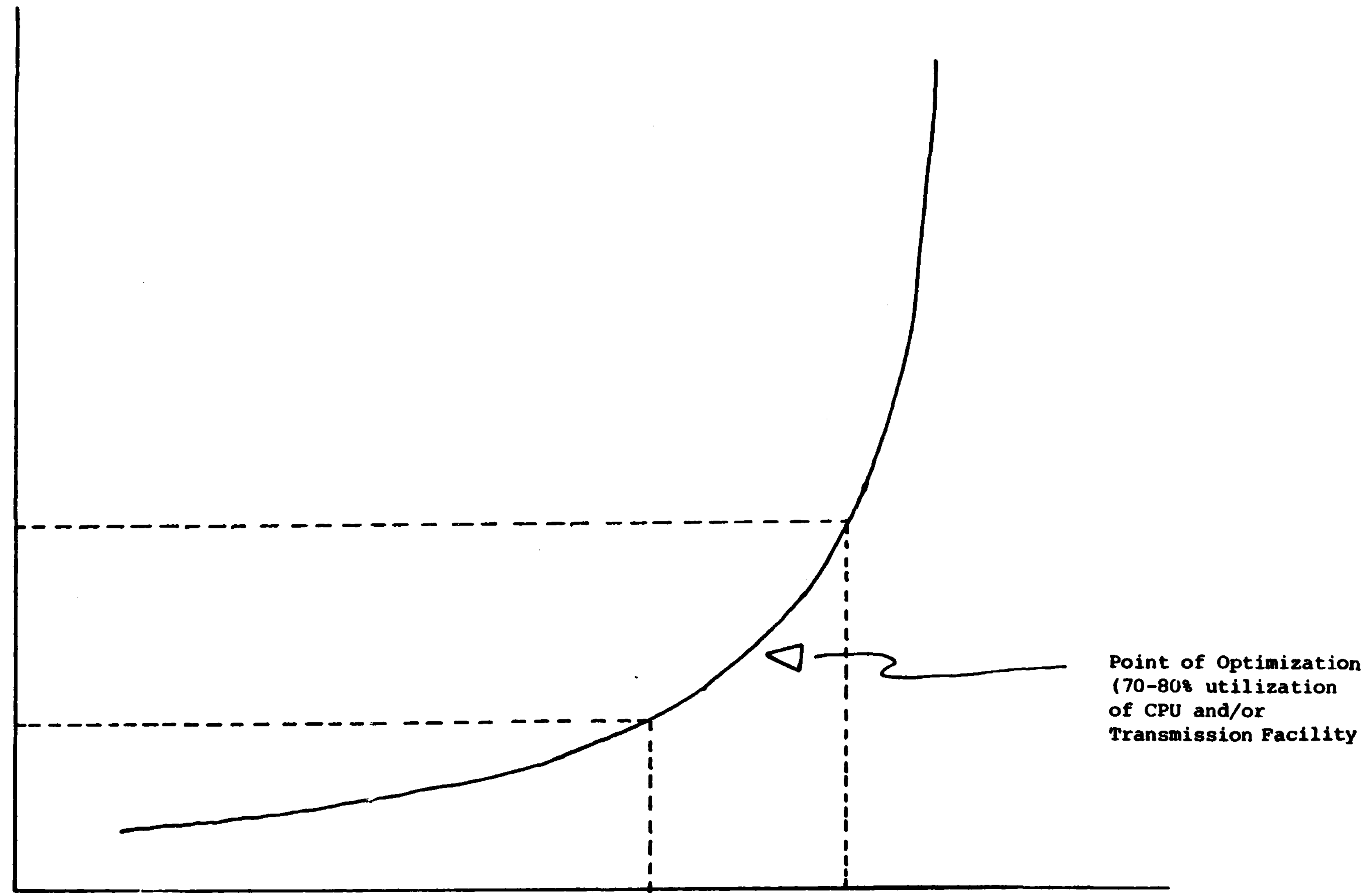


Figure 3-1
Relative Response Time vs. Throughput
Comparison in Packet Switches

The use of concurrent processing techniques addresses the throughput requirements of the system. An AUTODIN II node consisting of an interface processor and two node processors (PDP-11/04,34) must be able to handle 250 KBPS in traffic. This environment indicates that an implementation of the SCI architecture must generate modules that execute efficiently and intermodule interactions must be rapid.

3.2 HARDWARE CONSIDERATIONS

In this section, we will justify our assumption of a wide range of hardware environments. Additionally, the management and distribution of hardware resources will be examined.

Communication software has proliferated across all hardware boundaries, including hardware type, size, architecture, and vendor boundaries.

The current trend, however, is to transfer communications-related overhead out of the larger mainframe environments. Communication software is essentially spreading out into the "channel" itself. Communication software is present in front-end processors, communication controllers/multiplexors, intelligent line controllers, intelligent terminals, and even smart transmission lines (microprocessor-based frequency/time division line multiplexors). Well-designed and implemented network systems comprised of mini- and micro-machines are capable of considerable sophistication and performance.

3.2.1 <u>Hardware Resources</u>

The primary hardware resources of a communication system are the CPU, memory, and the transmission facility access.

Access of the CPU and the transmission facility is potentially resolved via the software/hardware configuration and a scheduling algorithm that incorporates priority/demand/supply considerations.

Access of memory resources is not as straight forward. A design requirement is that memory that is not allocated to coding structures is made available to the system, at compile time, in the form of common buffer pools. Acquisition of portions of this fixed memory space represents a dynamic in-line acquisition by the requesting process.

Memory resources have expanded greatly with the strides made with memory technologies. However, communication systems have been and probably will continue to be memory-space constrained for the following reasons:

- User requirements as well as system requirements will continue to grow.
- Performance of communication systems is tightly coupled to the amount of buffer space available to the system.
- The number of users of a communication system is directly proportional to its success.
- Implementations are gravitating toward smaller environments.

Memory resources must be closely monitored and managed to ensure proper operation of such a system.

3.2.1.1 Single Processor Environments

Figure 3-2 represents this type of configuration. Contention for resources is strictly at the user (and subsequent interrupt) level and is resolved by the scheduling and IOC algorithms. Whichever user is being serviced, at any point in time, potentially has access to all the required resources.

3.2.1.2 Multicomputer Configurations

/BBNE79/ distinguishes multicomputer configurations, illustrated in Figure 3-3, from a general class of multiprocessor configurations represented in Figure 3-4. The distinction is the lack of shared memory between the processors of the configuration.

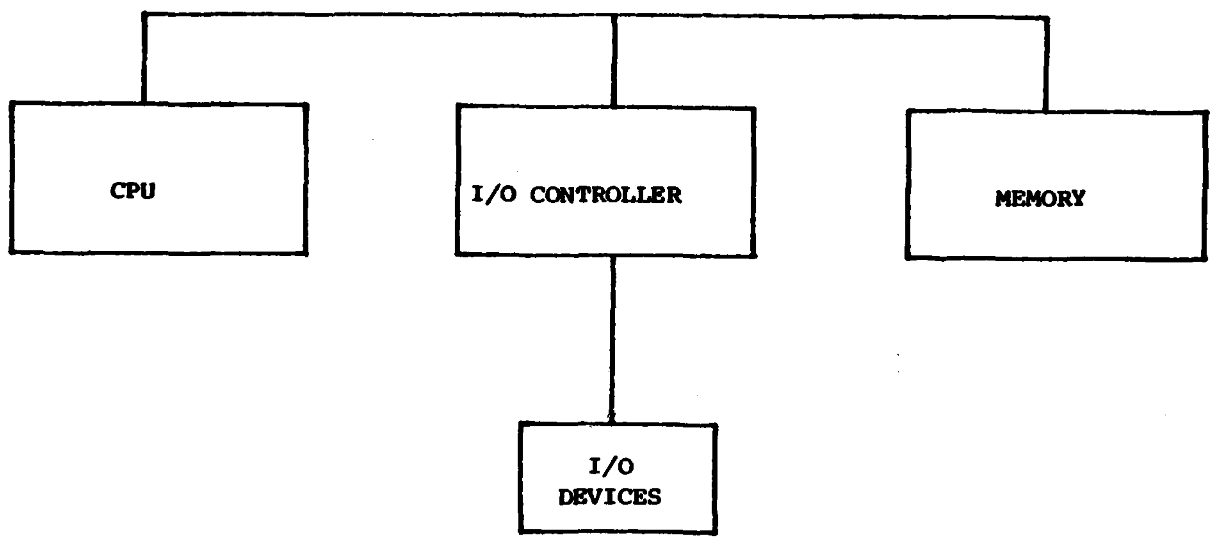


Figure 3-2

Single Processor System

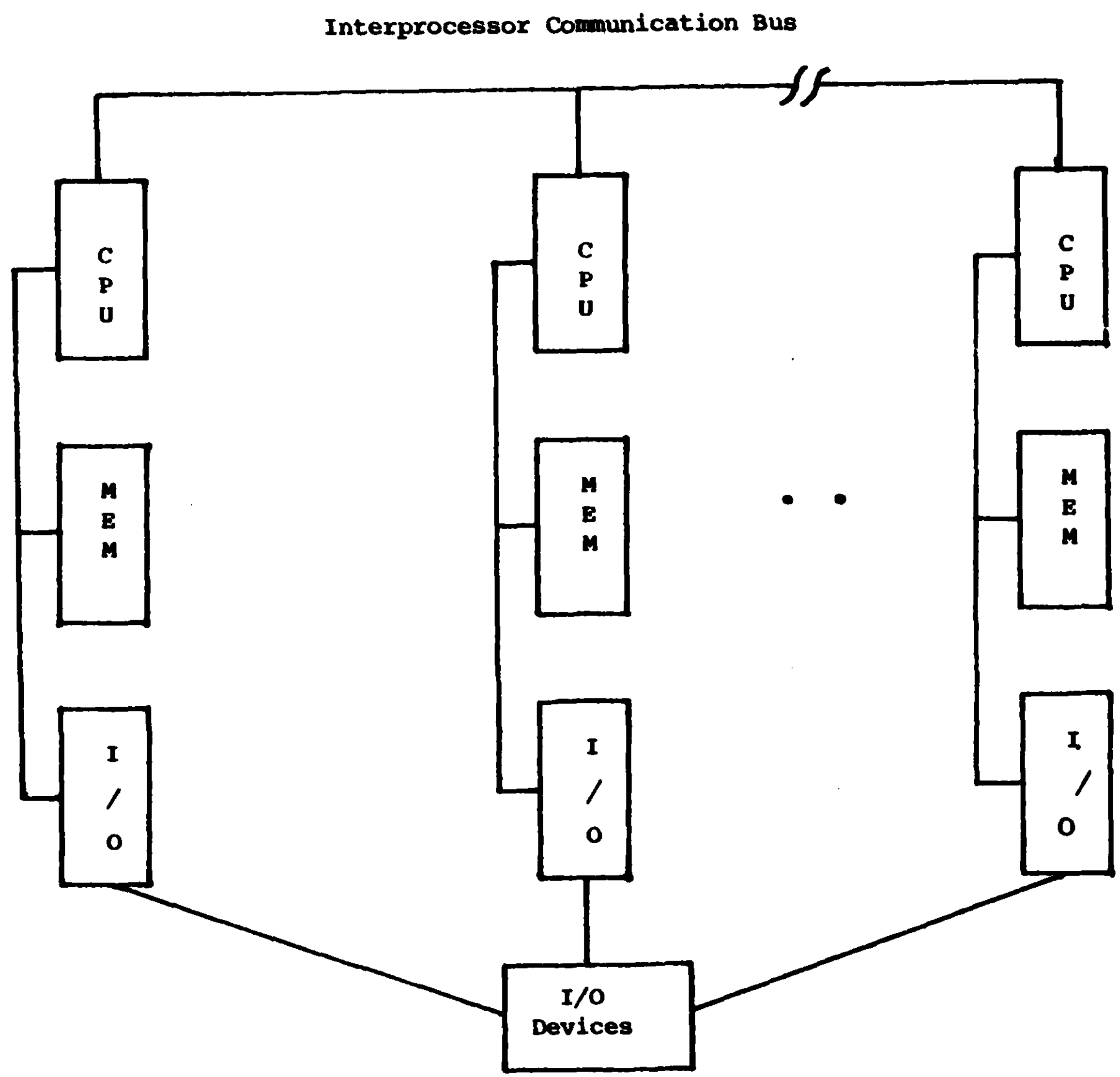


Figure 3-3
Processor Network Configuration

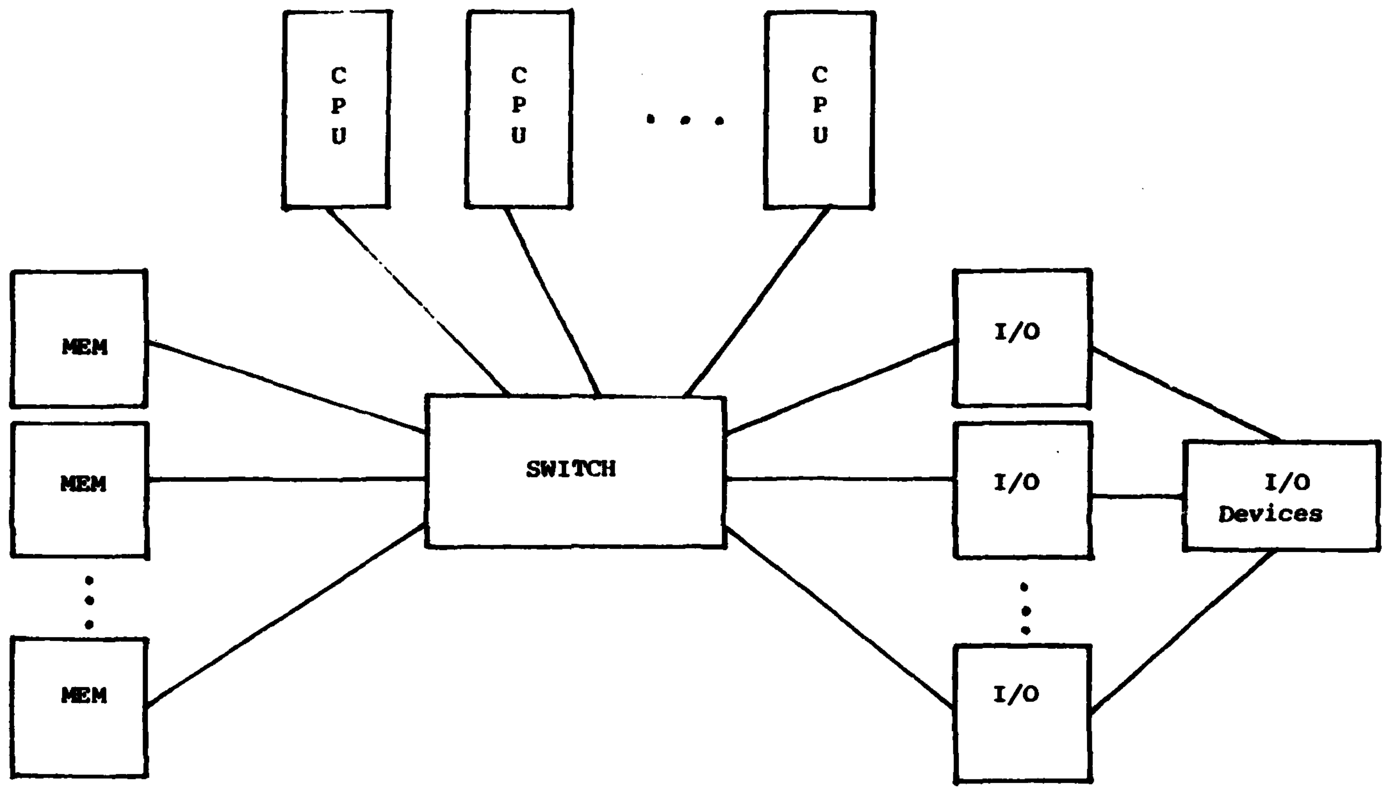


Figure 3-4

Fully Interconnected Multiprocessor System

This type of configuration is utilized to address increased throughput and connectivity requirements of an existing packet switch. The communications bus is utilized for local intra-node message traffic.

From the communications point of view, these configurations do not represent a quantum difference from a series of single processors interconnected via the network transmission facilities. Intra-node message traffic over the communications bus would be subject to peer layer or layer/layer interface conventions. (Although this method of intra-node communication is occasionally used, it is not an efficient or effective communication technique with regard to use of resources and response time.) Access to the bus would be according to local hardware conventions and legislated by system management and operating system software algorithms.

3.2.1.3 Multiprocessor Configurations

Using the descriptions in /BBNE79/, a multiprocessor is a multicomputer configuration with shared memory. Multiprocessors, illustrated in Figure 3-4, represent significant interprocess resource contention and communication potentials.

Multiprocessor environments form the basis for the concurrent processing issues that have been raised concerning high-level language implementations of communication systems. The protocol layers provide for widely distributed processes to synchronize and control data exchanges via transmission facilities. Locally distributed processes may have the same requirement to synchronize and exchange data; however, to use the full SCI architecture and the transmission facilities to achieve this would be a very inefficient use of multiprocessor facilities.

3.2.1.4 Future Considerations

The requirements of current and future communication systems hardware environments illustrate a greater dependence on concurrent processing capabilities. The following considerations illustrate this point:

- The use of multiprocessor configurations should increase to address throughput capacity, reliability, and connectivity requirements of the larger, highly interconnected systems.
- Traditionally, communication systems have been conversational, transaction-oriented bursts of high activity followed by longer periods of no activity. This mode of operation raises two problems: one consists of managing the transmission resources on a real-time demand basis; the other consists of overall inefficient use of the transmission facility. Communications systems should gravitate toward multi-access interconnection mechanisms. The ability to multiplex transmissions on a single transmission facility is currently accelerating. Fiber optics, laser, and microwave transmission technologies, their reduction in cost, and the refinement of random access protocols /DECO80/ will bring this about /TOBA80/. The consequence is that transmission requirements could approach (and possibly surpass) computer software capacity. This in turn could drive the implementation of standard protocols even further and faster into hardware/firmware structures to address future throughput requirements.
- Generally, more diverse (and specialized) users will require greater interconnection with each other.

- Communications systems will be utilized for digitized voice, image processing, and general machine-to-machine types of interactions.

3.3 ARCHITECTURAL CONSIDERATIONS

This section will deal with the issues associated with an Ada implementation of the SCI architecture. If an additional level of detail is applied to the SCI architecture, Figure 3-5 results. This drawing illustrates some of the more intrinsic and subtle relationships that can exist within the architecture. Adherence to the architectural model will require:

- Resolution of external interfaces.
- Modular structures.
- Common data structures.
- Internal organization of the architecture.
- Implementation of a scheduling algorithm.
- Access to a timing mechanism.
- Careful, complete definition of system requirements.

3.3.1 External Interfaces

The SCI architecture will address three external interfaces which consist of the Operating System/Executive Software, Communications Hardware, and User Interfaces.

3.3.1.1 Operating System/Executive Software Interface

The interface is determined by the operating system and may range from very low level memory manipulations to very high level, multiple parameter calls. It is this interface and the range of management services provided that achieve the operating system quality of communication systems software. Communications systems are implemented on a wide range of hardware environments. Consequently, they experience a variety of operating system/executive environments that range from having no explicit operating system to sophisticated multiprocessor operating systems.

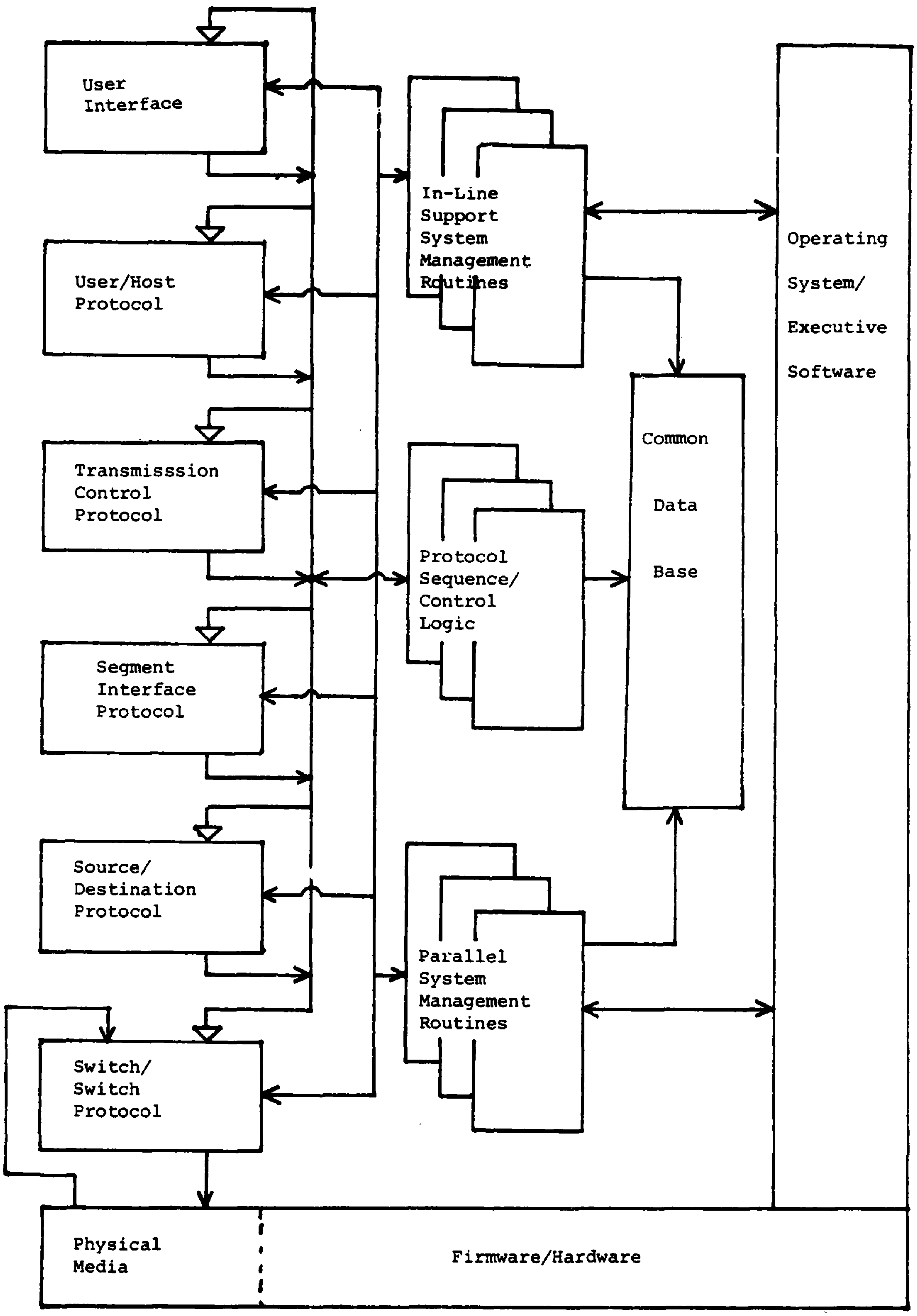


Figure 3-5
SCI System Architecture

These environments are not always favorable ones. As /CLAR80/ pointed out, the analysis of an implementation of the DoD model on a large MULTICS system illustrated that the operating system was responsible for most of the internal processing and that this overhead did not vary with the length of the data transfers. Operating system scheduling algorithms and memory management schemes may have to be dynamically, and quite drastically, altered or circumvented to provide a communication system with sufficient resources to operate at a required level of performance.

3.3.1.2 Communications Hardware

Communications equipment can vary widely in the areas of performance, complexity, and intelligence. This equipment actually falls on a continuous curve concerning the parameters just mentioned. The following paragraphs will highlight characteristics at the upper, middle, and lower sections of this curve.

3.3.1.2.1 High-Level Communications Devices

High level communication devices are software/firmware driven front-end or communication processors. Access to these devices is at a file level of I/O. The characteristics and subtleties of the communication process are shielded from the central processor. Considerable bandwidth over a number of different types of lines is possible. The I/O interface to these devices could be at the transmission control protocol level of the SCI model.

3.3.1.2.2 Mid-Level Communications Devices

These devices have considerable hardware/firmware complexity, but they remain directly under the control of software. I/O is at a message level or lower.

The interface in the SCI model is at the source/destination protocol level or lower. The following functions are typical:

- Enabling, disabling, and fielding of interrupts is performed.
- Allocation and release of data buffers is performed.
- Buffered I/O employing DMA is used.
- Supervisory/Control I/O exchanges are performed.
- Transmission line characteristics are addressed by the hardware/firmware.
- Medium-to-high line speeds are supported on a group of similar line types.

3.3.1.2.3 <u>Low-Level Communications Devices</u>

These devices have no sophistication. They are employed where lower costs and lower performance requirements prevail. I/O processes are at a very low level:

- The transmission line interface is manipulated directly by software.
- Interrupts are accepted, enabled, and disabled.
- Parity checking, redundancy checking is accomplished at the software level.
- Transfer is on a character-by-character basis.
- Low to mid-range line speeds are supported on a single line.

3.3.1.3 <u>User Interface</u>

In practice, this interface varies widely from system to system. Generally, the interface is determined by the communication system software; however, it will have to address the following conditions:

- Previously designed and installed software conventions.
- Human engineering requirements such as console and terminal operator display conventions.

- Mechanical/electrical requirements of low level devices.

3.3.2 Modular Structures

Ideally, communications software should consist of sets of functionally oriented modules (and supporting data structures) that can be dynamically linked and invoked according to real-time events and conditions. The SCI architecture is a significant step toward this end. Modules should have the following properties and characteristics:

- Be interruptible, re-entrant, and relocatable
- Be manageable in size with a common function orientation
- Share common data structures when needed
- Provide the ability to generate linkages to software written in other languages (native code, micro-code, assembly language, macro structures, and foreign language coded structures)
- Provide intermodule exchange of ID, control, and data parameters

3.3.3 Common Data Structures

Communication systems are event driven systems. An event can be associated with system state transitions. An efficient, flexible, and maintainable method of controlling software in this manner is to employ a data structure, similar to a common area in FORTRAN, in which to:

- Record event occurrences.
- Record the state of the system.
- Record the current component level user and hardware configurations.
- Record the current system level configuration.
- Record performance/utilization parameters.
- Share memory resources.

Extensive use of data structures provides a key advantage; when memory constraints exist, common data structures are a means to share resources and can be allocated and reclaimed dynamically.

3.3.4 Internal Organization of the Architecture

The architecture is partitioned into protocol layers and system management structures as referred to in Section 2. Thus, the need for a task-type software architecture is readily apparent. Conceptually, each protocol of the protocol layers can be viewed as being a task or a nesting of tasks that can be scheduled on a demand/priority basis.

3.3.5 Scheduling Considerations

In many respects, the software of communications systems behaves inherently as a message processing system itself with information, data, and control passing from one protocol layer to the next. Each such exchange represents a request for the use of one resource by another and thus represents a need for scheduling the use of that resource. Since many of these exchanges, both across a protocol interface as well as within a protocol layer, will be event driven, there exists a need for defining and being able to control parallel processes whether the implementation is in a single processor or multiprocessor. Thus, to accomplish this scheduling successfully, the SCI architecture relies upon a separate scheduling mechanism which is suited to the application requirements but which is implemented outside the protocol applications themselves as part of the operating system software of the SCI model.

3.3.5.1 Scheduling Criteria

Task scheduling is based on a multifaceted set of priorities. The SCI architecture dictates that events at the physical layer have a higher priority than at the user level. General efficiency dictates that resource-freeing processes

have priority over resource-acquiring processes. Static priorities are implemented via design considerations. Dynamic precedence priorities may need to be formulated and acted upon in real-time.

Flexible, efficient, comprehensive, and dynamic scheduling algorithms must be possible within an implementation of the SCI system.

The following states or events could invoke a scheduling operation:

- Task suspension criteria
  - Initiation of I/O (Optionally dependent upon the CPU/IOC hardware configuration)
  - Initiation of a time delay
  - I/O termination for a higher priority task
  - Empty message queues
  - Expiration of a time slice (Solely dependent upon the operating system)
  - Task termination
- Task activation criteria
  - Termination of I/O
  - Expiration of a time delay
  - Non-empty input queues
  - Time slice available

3.3.5.2 Scheduling Mechanism

A mechanism with sufficient comprehensiveness, flexibility, and efficiency is not perceived to be inherent within any high-level language or available within general operating system software used in communication systems. In addition, tasks should not be required to be directly involved in the scheduling function. Rather, tasks should be permitted to perform actions which result in scheduler interactions which in turn assure that the best use of available resources will be made. Evaluation of any implementation language must assess whether a scheduling algorithm can be implemented within the syntax and semantics of the language and according to a multifaceted set of scheduling conditions and priorities.

Generally, the scheduling of tasks would be provided by a system management scheduler, a multiprogramming/multitasking executive, or some combination of the two which determines which task can execute next and then invokes the operating system dispatch mechanism to handle the mechanics of the task context switching.

3.3.6 Timing Functions/Mechanisms

All protocols will require timing functions to varying degrees of accuracy. Timing of I/O processes, protocol segments, and institution of delays are essential functions of a communication system. Timing functions are utilized to delay processing until resources are available or specific conditions are met, to provide response timing windows to maintain protocols and interlayer interfaces, to detect idle conditions within the system, and to detect certain types of errors.

A mechanism for timing functions could be a message sent to the interval timing device software. Expiration of the interval is reported to the requesting process via a priority head-of-queue return of the message to the requestor's message queue.

3.4 SOFTWARE ENGINEERING CONSIDERATIONS

In this section, we will define software engineering as a formal, structured approach to management of the life cycle of a product. Phases of the product life cycle are:

- Requirements Definition
- Design
- Implementation
- Operation/Maintenance/Support

With the above background, areas were sought in which a high level language would significantly impact a general engineering approach to the implementation of communication system software. Thus, the major issue is how the Ada language can be of assistance early in the design phase of the product life.

Three tools associated with the design process have been identified that would lend themselves to Ada implementations:

- Protocol specifications and program language models /BOCH80/

The feasibility of using Ada for program language models is demonstrated in /BOCH80/; "program language models are motivated by the observation that protocols are simply algorithms, and high level languages provide a clear and relatively concise means of describing algorithms." This technique is demonstrated in /BBNE80/.

- Program design language

An Ada translation of the program design language description of the ADCCP protocol /AUTO78/ which is utilized by the AUTODIN II system, produced a highly understandable and improved documentation/specification vehicle.

- Simulation Languages

As /KOBA78/ points out, it is often desirable and preferable that simulation vehicles be implemented in general purpose high level languages to increase debugging potential, compiler support potential, and decrease multidisciplined, cross training, and multiple resource (simulator language) support.

3.5 SUMMARY

In this section, we have identified the environments and practices associated with communication systems. The following issues associated with using a high-level language as an implementation vehicle have been identified.

3.5.1 Performance Issues

Packet switch networks represent severe performance requirements. These requirements dictate that modules execute efficiently, and also that a choice of algorithms for process scheduling and interaction be available such that the most appropriate may be utilized.

3.5.2 Hardware Issues

Several hardware issues have also been identified: first, the implementation of communication systems software across all hardware boundaries implies transportability of the software across a broad hardware range. This will require wide acceptance of the language and/or very comprehensive cross compilation capabilities. Second, the management of access to resources in the single processor and multiprocessor environments must be efficiently resolved. Third, memory space constraints on communication systems require that modules coded in a high-level language compile into efficient machine code. Fourth, the use of smaller environments for communication systems implies (1) the ability of the software to fragment and execute across architecture/vendor classes of hardware, and (2) the ability of the language to serve as an implementation vehicle or guide to lower level, less sophisticated hardware units. Fifth, efficient interfaces to mid/low-level hardware devices must be possible.

3.5.3 Architectural Issues

The following architectural issues have been identified:

- Varying Operating System Environments
  Communication systems are implemented in a wide range of hardware and associated operating system environments. High level language constructs should be sufficiently flexible to address this situation.
- Common Data Structures
  The SCI architecture utilizes common data structures to address memory space constraints.
- Efficient and Comprehensive Scheduling Vehicle
  The SCI architecture is predicated on an efficient, comprehensive, and real-time scheduling capability.

- **Access to Interval Timers**
  Communication systems require a suite of timing services.

SECTION 4
CONCURRENT PROCESSING CONSIDERATIONS

Concurrent processing capabilities in communication systems generally, and the SCI architecture in particular, partially address the time and space requirements of communication systems by sharing the primary resources of the system. This section of the document will define communication systems concurrency requirements, identify the various concurrent processing environments, identify concurrent process associations within the SCI architecture, and examine conventional approaches to process control. This treatment will yield the concurrent processing issues and problems of a high level language implementation of communication systems software for which solutions and alternative measures can be determined and analyzed.

4.1 COMMUNICATION SYSTEMS SOFTWARE REQUIREMENTS

Concurrent processing is experienced in the SCI architecture to address multiple users of the system, and the invocation of system management processes that operate in parallel or concurrent to the sequential protocol layers.

As Figure 3-1 illustrates, there is a significant throughput/response time tradeoff consideration exhibited in packet switch systems; that being an optimization on both parameters. Concurrent processing generally addresses the throughput parameters, while the sequential processing exhibited by the protocol layers addresses the response time parameters.

4.2 CONCURRENT PROCESSING ENVIRONMENTS

The degree of concurrent processing and process control that exists within a system is dependent upon the system requirements, the set of management services/functions provided, and the hardware environment. This section will

examine the concurrent processing requirements of communication systems in the light of various processor configurations.

4.2.1 Single Processor/Multicomputer Environments

Single processor environments, as illustrated in Figure 3-2, are multiprogrammed, multitasked, or time-shared environments. As a result, true parallelism or concurrency is not achieved. However, the threads of control and process interaction proceed as if true parallelism were possible. Process-to-process communication is via the exchange of messages and process-to-process synchronization and processor resource contention is handled by the scheduling algorithm. Multicomputer configurations, as illustrated in Figure 3-3, are not a significant departure from the single processor environment. This type of configuration is usually implemented to address throughput and internode connectivity considerations.

Multitasking/multiprogramming in these environments exists to provide the capability for multiple system users and to invoke background (parallel) system management functions.

4.2.1.1 Multiple Users

Multiple users are interleaved via the SCI architecture. Figure 4-1 illustrates possible connection points (or delay/queue points) of a system employing several protocols at individual protocol layers.

4.2.1.2 Parallel Processes

The parallel processing considerations in these configurations are lower priority, system management types of processes. Again, true parallelism is not possible in these configurations. However, when conditions exist and resources are available, these processes will be scheduled at a lower priority, and the appearance of parallel operation results.

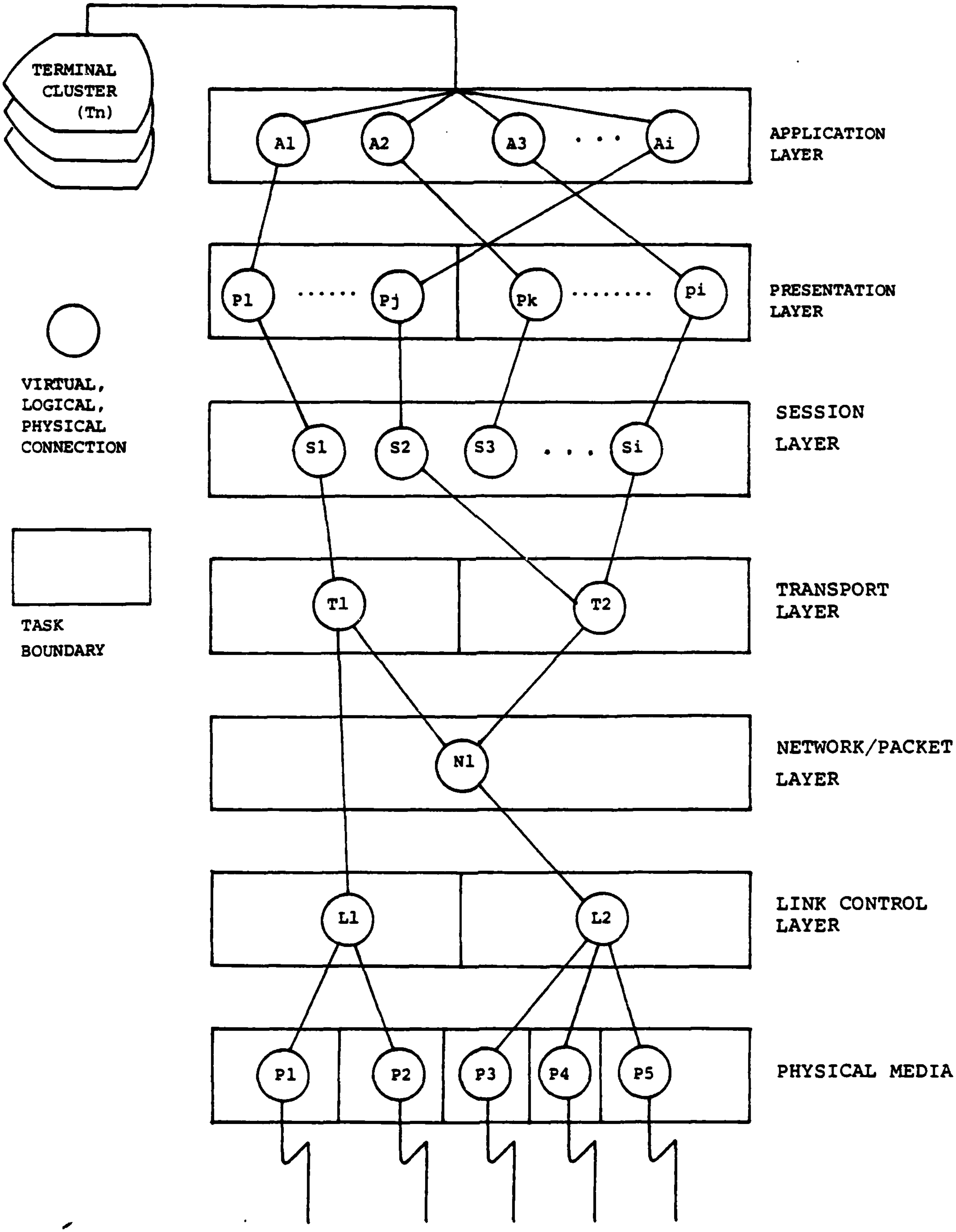


FIGURE 4-1
Multiple User
Connection Points Within the Architecture

4.2.2 Multiprocessor Environments

These configurations, as illustrated in Figure 3-5, represent a complete interconnectivity of the system resources (CPU, memory, and transmission facilities). True concurrent processing is possible, due to the sharing of memory between processors, to address the requirements on communication systems software of multiple users of the system and parallel system management functions.

4.2.2.1 Impact of Multiprocessor Configurations Upon the SCI Architecture

The SCI architecture is oriented toward multiprogramming/multitasking environments to address the event driven nature of the software and the requirements of servicing multiple users and providing parallel system management types of functions and services. As /JONE80/ points out, "... there are few differences between multiprogrammed systems with and without multiprocessors." The differences are perceived to be the mechanics of process control, which should be for the most part transparent to the protocol layers of the architecture. Thus, the solutions to the issues of associated concurrent processing and process control lie in the implementation of system management processes and structures. It is for this reason that significant discussion concerning the OSI and DoD models was presented and a prototype model, the SCI architecture, was developed. It is our contention that the SCI architecture is preserved in multiprocessor configurations; and any implementation is facilitated by the architecture concerning fragmentation and duplication of the architecture across multiprocessor configurations.

4.2.2.1.1 Sequential Processes in Separate Machines

Shared memory greatly facilitates the fragmentation of the SCI architecture along protocol layer boundaries as illustrated in the AUTODIN II system implementation. Such fragmentation requires the following restrictions. First, the hardware boundary must coincide with

the software boundary. Second, if a hardware interface is employed at the boundary (such as a bus or I/O channel), then the interface must be efficient and comprehensive (perform error checking, flow control, etc.). Third, if the interface between machines is via common memory, then mutual exclusion of the associated data structures must be enforced. Sequential processes in separate machines may be of the logically associated type, such as adjacent protocol layer tasks; or they may be of the logically connected type, such as a buffer manager routine and the task requesting the resource.

4.2.2.1.2 Parallel Processes in Separate Machines

These types of processes may be either logically associated, connected, or disjoint. Logically disjoint processes require no knowledge of each other and no control with regard to each other. Logically associated processes (such as peer protocol layers) require communication and mutual exclusion legislation but do not require synchronization in time.

4.3 CONCURRENT PROCESSING ASSOCIATIONS

Communication systems are event driven systems. At each layer of the SCI architecture, random message events, protocol events, timing events, and sequence considerations are sensed and acted upon, which illustrate a nondeterminant type of processing.

Concurrent processing associations in communication systems software can be categorized as follows:

- Disjoint processes
- Associated processes
- Connected Processes

4.3.1 Disjoint Processes

These processes do not require knowledge of or dependence on each other to complete their operation. The only commonality between disjoint processes is system time, memory space, and I/O resources. Disjoint processes are exhibited in

the SCI architecture as system management processes that execute in parallel to the protocol layer processors. Figure 4-2 serves as an illustration.

4.3.2 Associated Processes

These processes may require knowledge of each other and a degree of loose dependence upon one another. This type of association is exercised via interprocess communication and possible sharing of a common resource such as a queue strictly as producers or consumers, distributed in time. The concept of associated processes is a departure from the literature in a strict sense. /ICHB79b/ states "one of the important concepts introduced by /CONW63/ ...is that synchronization and data transmission are two inseparable activities". In communication systems, process-to-process communication is a necessary condition for synchronization; it is not in all instances a sufficient condition. For example, the sending of a message across an interlayer interface requires the sending task to wait only for the length of time necessary to deposit the message on the receiver's input queue. There is no requirement to wait for the receiver to accept or act on the message; in fact, to minimize the response time parameter, it is undesirable to do so.

Perhaps the inconsistency arises due to lack of a time parameter. The SCI architecture is designed to forward messages between processes in an asynchronous, user/server, relationship and distributed in time. Events at the application layer are measured on a completely different time scale than events at the lower layers. Synchronization (when required) is achieved by the internal workings of the sending and receiving processes themselves via peer protocols and the protocol layer interfaces. Figure 4-3 serves to illustrate logically associated interprocess relationships.

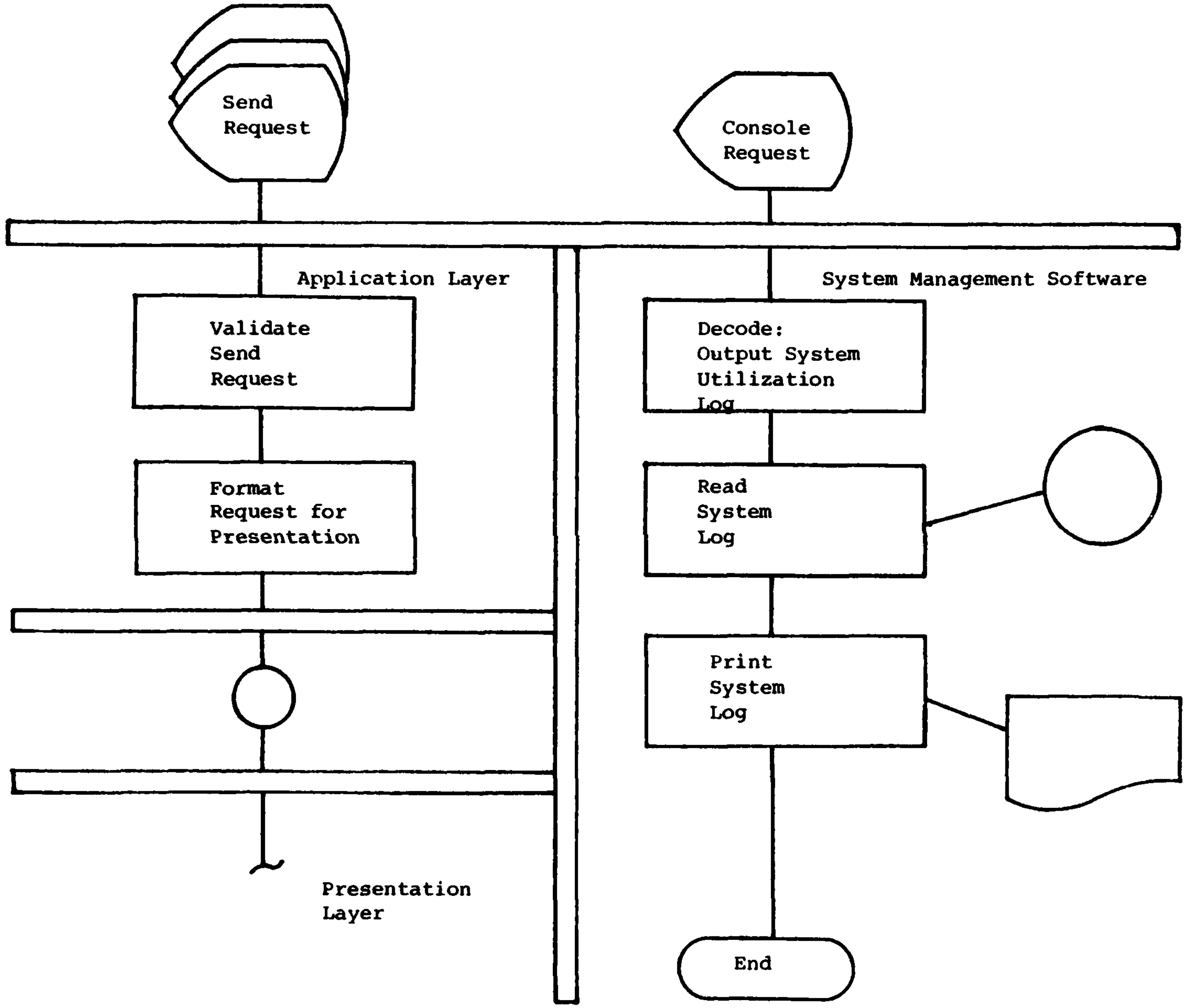


Figure 4-2

Logically Disjoint Processes

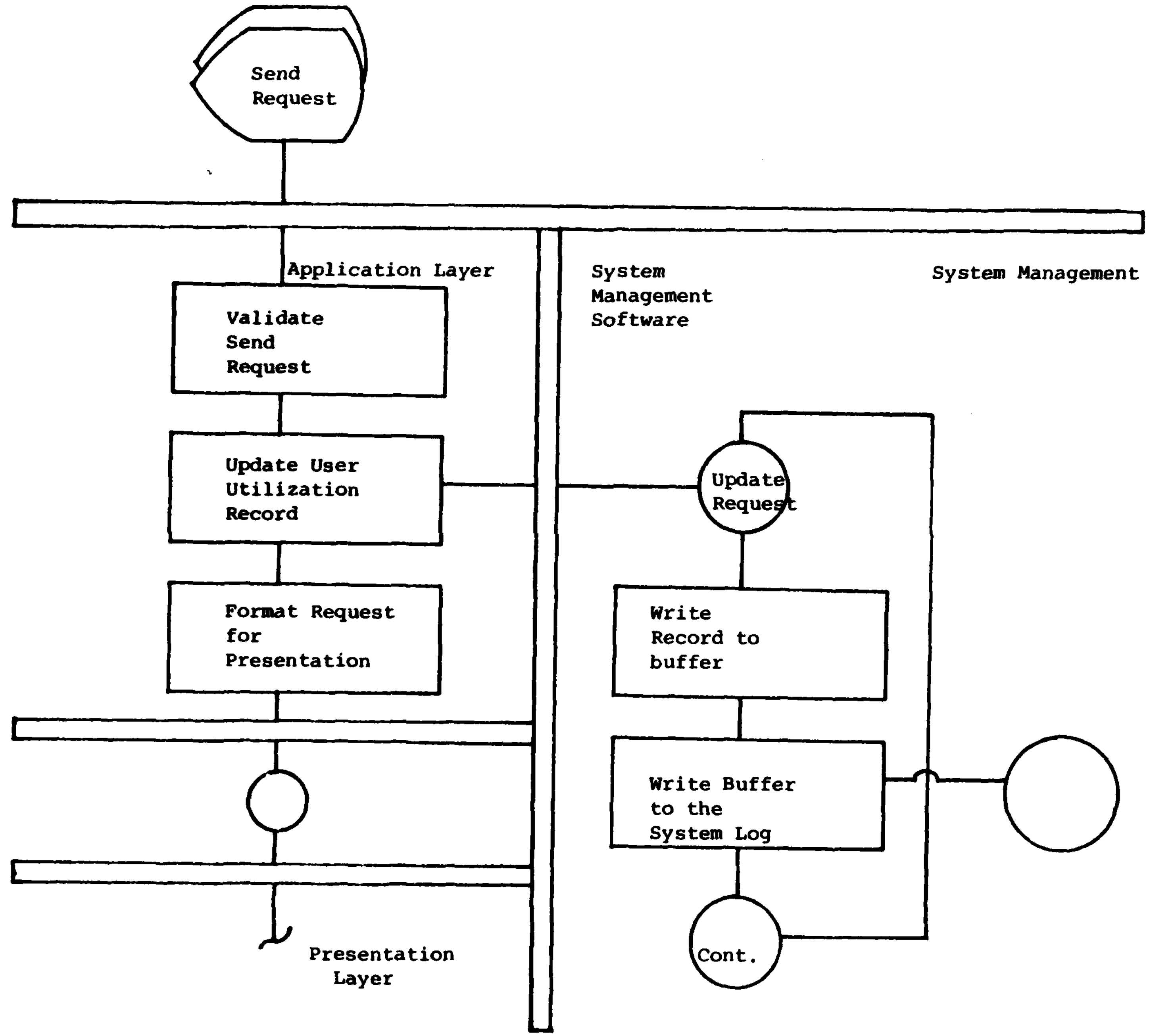


Figure 4-3
Logically Associated Processes

4.3.3 Logically Connected Processes

These processes require, at some point in time, and possibly space, knowledge of and a high degree of dependence upon one another to complete a function. Typically, a process cannot complete, or proceed to the next step until the operation of one or more logically connected processes have completed. Logically connected processes exercise this type of association via communication with one another, synchronization with one another in time, sharing of common resources in a producer/consumer dependence or, in producer/producer, consumer/consumer contention. Figure 4-4 illustrates this type of relationship.

4.4 TRADITIONAL SOLUTIONS TO PROCESS CONTROL

The following subsection defines those facilities available within current programming languages that are used to support process control. Process control in this context may be viewed as those commonly known concurrency aspects of process-to-process synchronization, process-to-process communication, and mutual exclusion. Each of the mechanisms outlined below is used to support one or more of these aspects. Advantages and disadvantages associated with the usage of these mechanisms are also presented.

Section 4.4.2 presents a description of how these traditional solutions to process control apply to general communication system implementations.

4.4.1 Process Control Mechanisms

4.4.1.1 Interlocks

An interlock is a primitive and efficient mechanism used to provide access control to code or data segments within a program. It is normally implemented via a "TEST-and-SET FLAG" instruction in cooperation with hardware features that guarantee uninterruptible fetch and store operations on the flag in use.

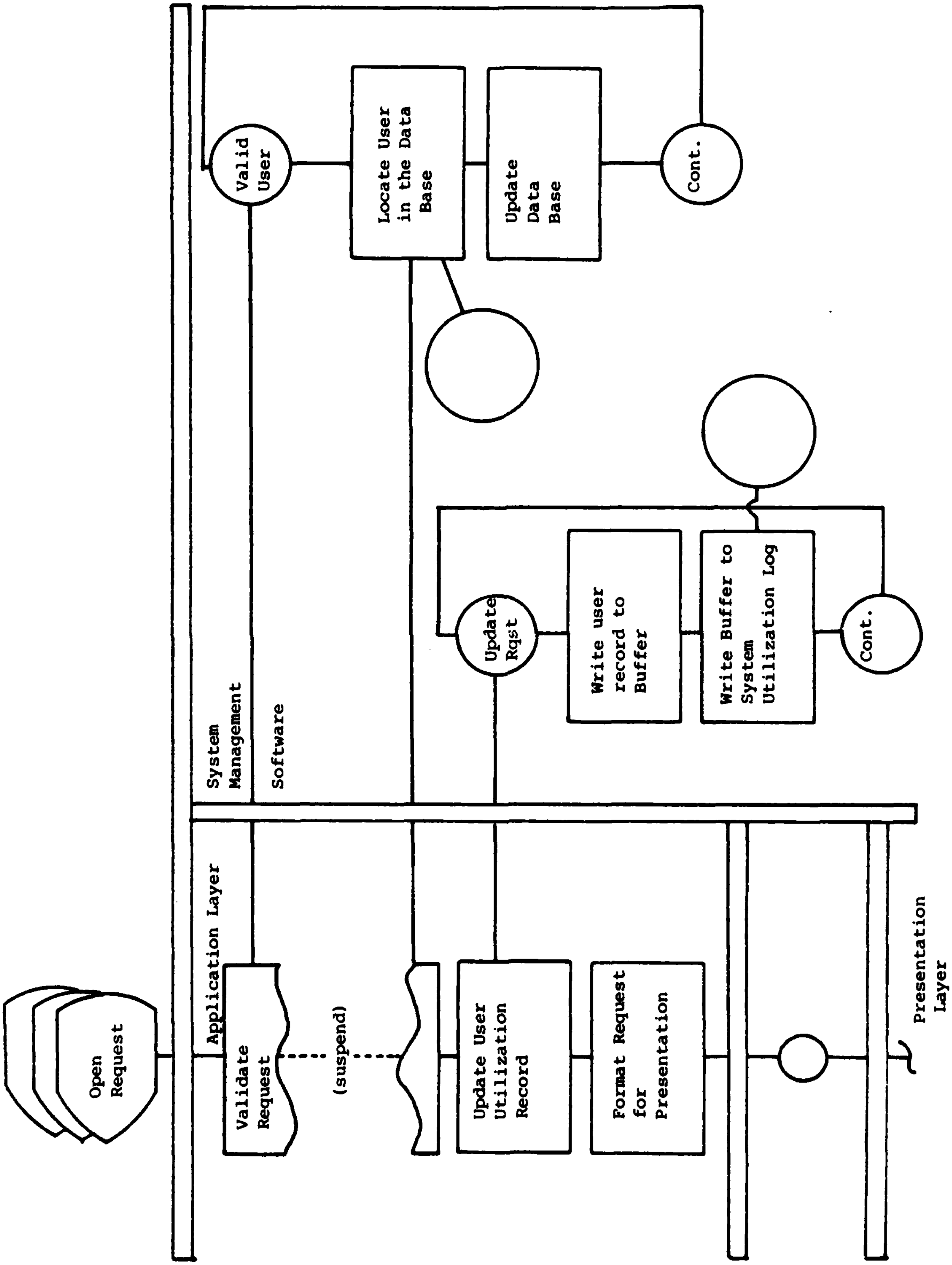


Figure 4-4
Logically Connected Processes

A typical implementation would include objects that could be LOCKED and UNLOCKED. For example, a programming language would typically declare an interlock variable (I) upon which LOCK and UNLOCK operations would be performed. When mutually exclusive access to a shared object is desired, this object would be surrounded by LOCK (I) and UNLOCK (I) primitives. That is, before using the object, a program should LOCK its corresponding interlock, and afterwards should UNLOCK it. Thus, mutual exclusion is achieved.

The advantages and disadvantages of employing interlocks as process control mechanisms are as follows:

<u>Advantages</u>

- Efficient implementation
- Simplicity of use

<u>Disadvantages</u>

- Lack of monitoring capability
- Lack of enforcement of access adherence or compliance
- Tendency to make program sections in which they appear difficult to maintain
- Tendency to defeat modular structure of surrounding program elements
- Only addresses mutual exclusion aspect of process control

4.4.1.2 Semaphores

Like the interlock, a semaphore is a primitive and efficient process control mechanism. It is often used in process control when a process is only concerned with receiving a timing signal from another process when a certain event has occurred, or when mutually exclusive access to a shared object is desired. It can be regarded as a special case of process communication in which an "empty message" is sent each time a ce-tain event occurs. Since the messages are empty, it is sufficient to count them and hence the semaphore (S) may be viewed as a single element buffer containing the number of signals sent, but not yet received (/BRIN73/).

The only valid operations defined on semaphores are P(S) and V(S), sometimes called WAIT and SIGNAL, respectively. These two operations allow a process to block itself to "wait" for a certain event and then to be awakened by a "signal" from another process when the event occurs. Thus, P(S) and V(S) have the following meaning:

P(S): Wait until $S > 0$, then $S = S-1$

V(S): $S = S+1$

Note that the operations P(S) and V(S) must exclude each other in time since the semaphore (S) is a common (shared) variable.

Semaphores exhibit many of the same advantages and disadvantages of interlocks.

Advantages

- Efficiency of implementation
- Simplicity of use
- Possess rudimentary scheduling potential
- Possess some access control

Disadvantages

- Lack of monitoring capability
- Do not rigidly enforce access adherence or compliance
- Tends to make surrounding program elements difficult to maintain
- Tends to defeat modular structure of surrounding program elements
- Only addresses synchronization and mutual exclusion aspects of process control

Note that the degenerate semaphore case, in which only the integer values 0 and 1 are employed, functionally corresponds to the interlock described previously. Such semaphores are called binary semaphores. Note also that the "critical section" or "critical region" /BRIN73/ syntactic form is really only a construct equivalent to a bracketed pair of P and V operations which prevents undesired entry and exit from the region and thus overcomes one of the most undesirable features of separately implemented semaphores. The conditional critical

region /BRIN73/ merely extends this to provide alt ative actions if a requested resource is busy.

4.4.1.3 Message Buffers

Interlocks and semaphores only address the synchronization and mutual exclusion aspects of process control. They cannot be used to directly effect message exchange between cooperating processes. However, an extension of the semaphore primitives allows them to become communication operations that provide both synchronization and data transmission. Usually, SEND and RECEIVE operations are defined by which one process executes SEND to pass a message and a second process accepts the information by executing RECEIVE. Since it is desirable that the sending process not be blocked awaiting acceptance of the message by the receiving process, most implementations support the declaration of a message queue or "mailbox."

The advantages and disadvantages of message buffer mechanisms should be obvious, but are listed below for completeness.

Advantages

- Fairly simple to use
- Does not adversely affect modularity
- Reasonably maintainable

Disadvantages

- Tends to be inefficient due to overhead associated with message transfers and queue manipulation
- Does not directly address mutual exclusion or process synchronization

4.4.1.4 Monitors

A monitor provides convenient facilities for guaranteeing mutual exclusion and for blocking and signaling processes. It is defined in /BRIN73/ as: "A common data structure and a set of meaningful operations on it that exclude one another in time and control the synchronization of

concurrent processes." A monitor may be viewed as a "fence around critical data". All sequences of statements that manipulate shared data are collected and moved inside this "fence". The "fence" has several gates, one corresponding to each sequence of statements. Each of the sequences thus form a special purpose procedure called an "entry." This means that all the critical sections for a particular set of shared data are collected into one place /HOLT78/.

It can easily be seen that whenever one of these entries is invoked, mutually exclusive access to the shared data is automatically provided. Furthermore, the enforcement of mutual exclusion is implicit --- the programmer need only invoke the entry --- the translator is responsible for generating code to guarantee mutual exclusion.

The advantages and disadvantages are as follows:

Advantages

- Does not adversely affect modularity
- Guarantees mutually exclusive data access
- Precedence and priority considerations can be provided
- Supports maintainability from a modularity point of view

Disadvantages

- Somewhat inefficient in comparison to semaphores and interlocks
- Only addresses mutual exclusion and synchronization --- not communication
- Somewhat difficult to use and hence maintain (from a complexity point of view)

4.4.2 Applicability of Traditional Solutions to Process Control Within Communication Systems Software

The traditional approaches to implementing process control mechanisms illustrates a tradeoff between modularity and efficiency. Communication system software, such as the SCI architecture, exhibits requirements for process control mechanisms that span the efficiency/modularity spectrum.

- **Interlocks**
  The use of shared data structures requires the rapid access, mutual exclusion benefit of this type of device.
- **Semaphores**
  The implementation of efficient access control (synchronization and mutual exclusion) to interval timing devices could well be served by this type of device.
- **Message Buffers**
  Communication systems inherently employ message exchanges between processes. Message queues or "mail boxes" serve the data exchange or interprocess communication requirements of the SCI architecture.
- **Monitors**
  The system management portion of the SCI architecture provides for the centralization of common monitor type, in-line services and functions (such as acquisition or release of a resource) for the protocol layers of the model. This type of device provides for mutual exclusion and strict compliance of access to system resources and shared data structures where required.

4.5 THE ADA LANGUAGE SOLUTION TO PROCESS CONTROL

The Ada language, as documented in /USDO80/, has addressed concurrent processing with the concept of tasks which can run in parallel with other tasks. The details of Ada tasking are documented in the literature (/USDO80/, /BBNE79/, /BOUT80/ and /ICHB79b/), and need not be repeated here in detail. The major concepts and points we wish to address are detailed below.

4.5.1 Ada Task Structure

The task structure which includes the entry, select, delay, and accept statements make up a formidable high level structure that maps well with the SCI architecture tasking requirements. The entry statement provides visibility to other processes; and it defines a queue for the calling processes. The accept statement addresses the retrieval of information (control and data) from the task requesting service. The select statement provides for an examination of a series of conditions (or guards) which together with input information determined the control of processing within the task. The delay statement provides for one of the timing functions required by the SCI architecture.

4.5.2 Ada Rendezvous

Ada uses the concept of a "rendezvous" between tasks to address process control for intertask communication, synchronization, and mutual exclusion. The characteristics of the task rendezvous are as follows:

- Asymmetry of Identity
  The calling task has knowledge of the called task. The called task has no knowledge of the caller, except possibly via the data that is exchanged, outside of a rendezvous.
- User/Server Connotation
  The called task acts as a server to calling tasks. Functions/processes are invoked by the called task on behalf of the caller.
- Scheduling
  The calling task is suspended until the rendezvous is complete. The called task is scheduled for execution, if not already executing, at the start of rendezvous (coincidence of an entry call and the execution of an accept statement). The called task continues executing for the duration of the rendezvous. The called task may be

suspended at the completion of the rendezvous and the calling task is rescheduled for execution.

- Queuing

  Ada associates a queue with each entry point in a task. This is the means for synchronization between tasks. Ada treats tasks as objects and within the framework of an Ada rendezvous, tasks are queued to one another.

NOTE: The /USD080/ does not specify what suspension means. Task suspension is an implementation decision and could conceivably be a spin-lock, a time delay, or a complete memory rollout of tasks.

4.5.2.1 Task Synchronization

Tasks are synchronized with each other within the task rendezvous of Ada. The calling task cannot proceed until the called task has completed, i.e., the accept statement has executed.

4.5.2.2 Task Communication

The exchange of data parameters between tasks may occur within the rendezvous. The mechanism is implementor/translator dependent.

4.5.2.3 Mutual Exclusion

Mutual exclusion of shared resources is achieved within the rendezvous since only one of the task pairs is actively processing until the rendezvous is completed.

No other mechanism is inherent within Ada constructs to effect mutual exclusion, outside of the rendezvous between task pairs.

4.5.3 Summary

The task structure and rendezvous concept chosen by the Ada designers generally provides for a high-level solution to process control. The Ada rendezvous concept has greater documentation and modularity potential than the traditional solutions discussed, however, it lacks flexibility and the efficiency of the more primitive mechanisms. The concept of rendezvous in theory maps very closely to the requirements, architectures, and overall purpose of communication system software.

Although not directly stated above, we believe that the other solutions to process control could be implemented via Ada constructs as modularity and efficiency requirements dictate.

SECTION 5
PROBLEMS AND ALTERNATIVES

This section addresses three categories of problems associated with Ada's ability to support communication software development. The first category stems from issues raised by BBN Report No. 4188 /BBNE79/ and concerns Ada's general ability to support concurrent programming activities. These issues are repeated and discussed herein for ease of reference. Alternatives or solutions to the stated problems are presented, where appropriate. The next category addresses issues specifically related to Ada's support of concurrency in a communication system environment. These issues stem from a mapping of Ada's concurrency facility onto the communication model developed in Sections 2 and 3, and specifically address how well this mapping compares with the required facilities described in Section 4. Once again, solutions and/or alternatives are discussed. The final category deals with miscellaneous other communication-related software issues where Ada exhibits problems or deficiencies. These other issues are included for completeness, even though the emphasis of this analysis effort is in the area of concurrent programming support requirements within a communication environment.

5.1 ISSUES RAISED BY BBN REPORT NO. 4188

BBN Report No. 4188, titled "The Impact of Multiprocessor Technology on High-Level Language Design", surveys several representative multiprocessor systems, describes classical approaches to process control and concurrency, and then evaluates the parallel control facilities provided by the Ada language in order to assess the practicality of using Ada as a standard language for existing multiprocessor systems. It should be noted that this report was published 10 September 1979, and, as such, only addresses the preliminary Ada definition /ICHB79a/, not that which is defined within the Ada Language Reference Manual (LRM) /USDO80/.

In the course of their evaluation, the authors raised several issues related to Ada's ability to support parallel processing within an assumed (generic) multiprocessor environment. These issues are presented and discussed herein.

5.1.1 Excessive Scheduler Interactions

5.1.1.1 Statement of Problem

The authors feel that run-time efficiency is the highest priority consideration in multiprocessor system implementations. As such, they were particularly concerned with evaluating Ada's parallel control features from an efficiency standpoint. Based on their evaluation, they concluded that the most severe problem with the process control features in Ada (from the point of view of efficiency) is that the transmission of data from a sender process to a receiving process requires excessive scheduler interactions.

In particular, they state, "The use of a complete rendezvous system results in unnecessary scheduling delays. This problem is particularly severe in the relatively important case of message passing in that Ada requires the sender of a message to wait for the scheduler before it is allowed to proceed."

This conclusion is based on certain assumptions as to the environment and the processes involved. The assumed environment is a single processor executing parallel processes in a message passing application. The processes involved are a sender process generating messages and entering these messages into a queue, and a receiving process which removes messages from the queue. READ and WRITE entries to a buffering task accomplish the message transfers. The authors contend that the scheduling problems arise from the semantics of the Ada ENTRY call issued by the sender process whereby the sender is blocked until the buffer task is scheduled and completes the rendezvous. During this time, the sender process is suspended and must wait to be rescheduled when the buffer task completes. The same basic sequence takes place when the

consumer task executes the corresponding entry call. Thus a total of four scheduling interactions are potentially required in this situation to transmit a single message. Also, since each scheduler interaction may involve a complete context swap, this implementation of message passing is considered to be prohibitively expensive for many applications.

5.1.1.2 Alternatives

When the BBN report was published, the inefficiency of Ada's tasking facility was a subject receiving considerable attention from the various language reviewers and the academic community in general. It is unfortunate that so much emphasis was placed on the inefficiency of Ada's rendezvous mechanism and so little emphasis placed on its advantages. It should be pointed out that a conscious effort was made by the language defining groups to avoid the proliferation of (the potentially more efficient) parallel process control constructs, i.e., the previously described (Section 4) interlocks, semaphores, etc. A trade-off exists between the efficiency of various constructs and their usability, implementability, reliability, and maintainability. While these lower level primitives are more efficient in their implementation, they tend to make the program elements in which they exist more difficult to correctly implement, less reliable in operation, and harder to maintain. An argument can be made that a certain percentage of real-time (communication) applications exist wherein the efficiency of the tasking facility becomes a problem. However, all of these applications must be highly reliable and easily maintained. The desire for a language to be efficient in operation often seems in conflict with the goals of expressive power and program clarity. Inevitably, trade-offs must be made, and hence the decision on which approach to use depends to a large degree on design priorities. The Ada rendezvous mechanism has obviously prioritized expressive power and program clarity in an attempt to foster the important goals of reliability and maintainability. While all of this presents a valid defense of

Ada's concurrency facilities, it falls short of offering legitimate alternatives in those situations where efficiency of implementation is a prime concern.

The first observation that can be made in dealing with this problem is that there is no direct alternative mechanism within the Ada framework which provides a more efficient implementation than the one described within the BBN report using the task rendezvous. If one implements buffered message passing with non-blocking senders in the manner described in the BBN report, one has to accept the inherent "side-effects" of Ada's task rendezvous mechanism and, in fact, it is readily agreed that potential difficulties can arise in certain applications where efficiency is a prime concern. One therefore has to search for alternatives to the problem rather than alternative implementations of the Ada rendezvous. In other words, the real problem lies not in making the rendezvous more efficient for this implementation but lies instead in the determination of an efficient alternative method of effecting message transfers between concurrently executing producer and consumer tasks in single processor, processor network, and multiprocessor environments.

With this in mind, an alternative based on manipulation of a common message queue is offered. Mutually exclusive access to the queue is provided by the inclusion of an interlock variable which can be locked and unlocked by the appropriate task. Example 5-1 shows a typical implementation. It can be noted that the message packets and associated control variables are defined in the same manner as in the BBN report example. The major difference is in the mutual exclusion provided by the interlock variable and the absence of the explicit task rendezvous for effecting message transfers. This alternative is in keeping with more traditional implementations of bounded buffer operations. The example as shown is oriented towards a single processor environment but obvious variations extend the concept to the multicomputer and multiprocessor environments, as well.

```
package MESSAGE is
  PACKET_SIZE:constant INTEGER:=256;
  type PACKET is array (1..PACKET_SIZE) of CHARACTER;
  SIZE:constant INTEGER:=10;
  BUF:array (1..SIZE) of PACKET;
  INX,OUTX:INTEGER range 1..SIZE:=1;
  COUNT:INTEGER range 0..SIZE:=0;
  type INTERLOCK is (LOCKED,UNLOCKED);
  type ACCESS_I is access INTERLOCK;
  L:ACCESS_I:=new INTERLOCK (UNLOCKED);
  procedure LOCK (L:ACCESS_I);
  procedure UNLOCK (L:ACCESS_I);
end MESSAGE;

package body MESSAGE is
  function TESTANDSET (L:ACCESS_I) return BOOLEAN is
     -- body of TESTANDSET function
  end TESTANDSET;
  procedure LOCK (L:ACCESS_I) is
     -- body of LOCK procedure
  end LOCK;
  procedure UNLOCK (L:ACCESS_I) is
     -- body of UNLOCK procedure
  end UNLOCK;
end MESSAGE;
     .
     .
     .
with MESSAGE; use MESSAGE;
package PRODUCER_CONSUMER is
  task PRODUCER;
  task CONSUMER;
end PRODUCER_CONSUMER;
```

Example 5-1 (Page 1 of 2)

```
package body PRODUCER_CONSUMER is
  task body PRODUCER is
  MSG1:PACKET;
  begin
     loop
        while COUNT=SIZE loop
           null;
        end loop;
        -- perform necessary processing
        -- to create desired message in MSG1
             .
             .
             .
        LOCK(L);
        BUF(INX):=MSG1;
        INX:=INX mod SIZE+1;
        COUNT:=COUNT+1;
        UNLOCK(L);
     end loop;
  end PRODUCER;

  task body CONSUMER is
  MSG2:PACKET;
  begin
     loop
        while COUNT=0 loop
           null;
        end loop;
        LOCK(L);
        MSG2:=BUF(OUTX);
        OUTX:=OUTX mod SIZE+1;
        COUNT:=COUNT-1;
        UNLOCK(L);
        -- perform necessary processing
        -- on received message in MSG2
     end loop;
  end CONSUMER;
end PRODUCER_CONSUMER;
```

Example 5-1 (Page 2 of 2)

In the example given, a producer task wishing to transmit a message to a consumer task enters a message on the queue only when the queue is not full and the interlock variable is unlocked, i.e., no other process is manipulating the queue. The details of the LOCK and UNLOCK procedures, and their associated interaction with a TEST and SET function, are given in Section 5.1.2.2. For now, assume mutually exclusive access to the queue is guaranteed through bracketed LOCK and UNLOCK procedure calls. As a producer task enters a message in the queue, it also adjusts the queue input pointer and increments the count of messages in the queue. As a consumer task removes a message from the queue, it likewise adjusts the queue output pointer and decrements the count of messages in the queue. Deadlock between producer and consumer tasks is prevented by checking for "queue empty" and "queue full" conditions prior to locking the interlock variable.

It can be seen that, while this example does not offer a more efficient rendezvous mechanism, it does provide a more efficient solution to the stated problem --- that of implementing buffered message passing with non-blocking senders using Ada constructs.

5.1.2 Process Control Structure Flexibility

5.1.2.1 Statement of Problem

The BBN Report maintains that Ada does not provide sufficient flexibility in its process control structure to allow a programmer to choose the mechanism which is most appropriate for the requirements of the application. The authors state ... "In Ada, the only mechanism available for providing mutual exclusion is through the rendezvous of an entry call in one task and an accept statement in another. Although we feel that the entry/accept linkage is a powerful tool which will be useful over a wide range of applications, there are limitations in the structure which will make it difficult to use Ada in certain applications environments in

which efficiency is of considerable importance unless additional primitives are included so as to provide a more flexible synchronization mechanism." Thus, the basic concern here is very similar to the previously stated problem. That is, if one assumes the rendezvous mechanism is an inefficient tool for synchronization, then Ada must provide other alternatives to the rendezvous mechanism (where appropriate) for certain application environments.

A second problem area relates to Ada's synchronization mechanism being control-based vice data-based. The argument here is that, in Ada, the entry/accept linkage results in the mutual exclusion mechanism being a function solely of the task (i.e., control structure) and not of the data structure (as in traditional implementations). The authors believe that the Ada control-based implementation leads to "convoluted program structures" or serious inefficiencies in the use of space.

5.1.2.2 Alternatives

The fact that Ada does not support a wide range of synchronization or mutual exclusion mechanisms was the expressed intent of the Ada design team.

In particular, on page 11-1 of the Ada Rationale /ICHB79b/ they state ... "One reason has clearly been a lack of confidence in the many different facilities put forward for the control of parallelism. Semaphores, events, signals, and other similar mechanisms are clearly at too low a level. Monitors, on the other hand, are not always easy to understand and, with their associated signals, perhaps seem to offer an unfortunate mix of high level and low level concepts. It is believed that Green [Ada] strikes a good balance by providing facilities which are not only easy to use directly, but can also be used as tools for the creation of mechanisms of different kinds."

Clearly, the Ada design team chose ease and consistency of implementation over a "grab bag" philosophy. This philosophy of one mechanism to handle all of the concurrent process control requirements is considered desirable

from a reliability and maintainability standpoint. Furthermore, this philosophy is not solely fostered by the Ada design team. Such notable experts in this field as Brinch Hansen and Hoare have proposed similar tasking implementations (/BRIN78/ and /HOAR78/) which strongly influenced the Ada design.

If, however, one desires to implement different mechanisms which could more closely address the requirements of a particular application, Ada provides the implementor with the flexibility to do so. The following examples show some of the ways Ada can be used to build other process-control mechanisms. These are by no means the only ways to implement these mechanisms but give an indication of the existing possibilities.

Example 5-2 illustrates an implementation of an interlock in Ada (interlocks were previously described in Section 4 along with the other "traditional" solutions to process control). In this example, a function TEST_AND_SET is defined by means of an assembly language routine which accesses the TEST_AND_SET instruction of the underlying machine. The example shows a typical AN/UYK-7 implementation. LOCK and UNLOCK procedures are then defined as shown. A call to the LOCK procedure will perform a busy wait operation until the function TEST_AND_SET returns a value FALSE indicating mutually exclusive access to a shared resource has been granted. A subsequent call to the UNLOCK procedure frees the resource for other users' access.

Ada can also be used to implement the traditional semaphore as shown by the following examples of binary and integer semaphores. The binary semaphore implementation shown in Example 5-3 was taken from the Ada Rationale /ICHB79b/.

A critical section of code performing mutually exclusive access to a shared data object can then be bracketed by successive P and V entry calls as shown in Example 5-4.

```
with INST_UYK_7;
function TESTANDSET(L:INTERLOCK)
                      return BOOLEAN is
  Q:BOOLEAN;
  procedure REAL_TESTANDSET;
  pragma INLINE(REAL_TESTANDSET);
  procedure REAL_TESTANDSET is
     use INST_UYK_7;
  begin
     FORM2'(OP= > TSF,A=> 0, B=> 0, I=> 0, SY=> L'ADDRESS);
     FORM3'(OP= > JNE,A=> 0, B=> 0, I=> 0, SY =>  LAB1);
     FORM1'(OP= > BZ,A=> 0, B=> 0, I=> 0, SY =>  Q'ADDRESS);
     FORM3'(OP= > RJ,A=> 0, B=> 0, I=> 0,
              SY = > REAL_TESTANDSET'RET_ADD);
   <<LAB1>>
     FORM1'(OP= > BS,A=> 0, B=> 0, I=> 0, SY = > Q'ADDRESS);
     FORM3'(OP= > RJ,A=> 0, B=> 0,
              SY =>REAL_TESTANDSET'RET_ADD);
  end REAL_TESTANDSET;
begin--function TESTANDSET
  REAL_TESTANDSET;
  return Q;
end TESTANDSET;

procedure LOCK(L:ACCESS_INTERLOCK) is
begin
  while TESTANDSET(L) loop
     null; -- do nothing (busy wait)
  end while;
end LOCK;

procedure UNLOCK(L:ACCESS_INTERLOCK) is
begin
  L.all:=UNLOCKED;
end UNLOCK;
```

Example 5-2

Note that in this case the rendezvous merely provides synchronization and no data are transferred. The P entry call acts as the traditional "WAIT on SEMAPHORE" action while the V entry corresponds to the "SEMAPHORE SIGNAL."

Similarly an integer semaphore may be implemented as shown in Example 5-5.

Again critical regions can be bracketed by P and V entry calls.

More elegant structures may also be constructed. For example, consider Example 5-6, a monitor implementation in Ada which illustrates one method of handling the classical readers/writers problem.

By maintaining a count of readers and the status of a writer and by being able to update these variables in a mutually exclusive manner, the READ_WRITE monitor in Example 5-6 ensures that readers never attempt to read while writers are modifying shared objects. As with the semaphore implementation, readers can bracket critical sections of code with READ_REQ and READ_REL entry calls and writers likewise with RITE_REQ and RITE_REL calls.

It should be noted that while the above examples provide means of implementing mechanisms more closely related to the intended application, the inherent disadvantages of these mechanisms (outlined in Section 4) are still present and should be taken into consideration during any implementation. It is felt that the above examples offer a range of "low level" facilities for mutual exclusion which adequately address the concern expressed by the authors of the BBN report. In applications where efficiency of implementation is not of a critical nature the normal utilization of Ada's task rendezvous mechanism as a means of providing mutual exclusion and synchronization is of course adequate and, in fact, desirable.

The second problem area, related to the storage inefficiency of Ada's control-based synchronization mechanism, can be handled quite easily in revised Ada. The solution lies in the ability to define "entity pointers" to objects of type ENTITY which contain a record with an INTERLOCK as its

```
task type SEMAPHORE is
   entry P;
   entry V;
end;
task body SEMAPHORE is
begin
   loop
      accept P;
      accept V;
   end loop;
end;
```

Example 5-3

```
LOC_SEM:SEMAPHORE;
      .
      .
      .
LOC_SEM.P;
   COMMON_DATA(TRACK_NUMBER):=TRK_INDEX;
   TRACK_NUMBER:=TRACK_NUMBER+1
LOC_SEM.V;
```

Example 5-4

```
task type INT_SEMAPHORE is
   entry P;
   entry V;
end INT_SEMAPHORE;

task body INT_SEMAPHORE is
   S:INTEGER range 0..INTEGER'LAST:=NUM_RESOURCE;
begin
   select
      when S>0=>
         accept P do
            S:=S-1;
         end P;
   or
         accept V do
            S:=S+1;
         end V;
   end select;
end INT_SEMAPHORE;
```

Example 5-5

```
task type READ_WRITE is
  entry READ_REQ;
  entry READ_REL;
  entry RITE_REQ;
  entry RITE_REL;
end READ_WRITE;
task body READ_WRITE is
  READ_COUNT:INTEGER range 0..INTEGER'LAST:=0;
  MODIFY:boolean:= FALSE;
begin
  loop
     select
        when MODIFY=FALSE and RITE_REQ'COUNT=0
           = >
           accept READ_REQ do
              READ_COUNT:=READ_COUNT+1;
           end READ_REQ;
     or
           accept READ_REL do
              if
                 READ_COUNT>0
              then
                 READ_COUNT:=READ_COUNT-1;
              end if;
           end READ_REL;
     or
           when READ_COUNT=0 and MODIFY=FALSE
           = >
           accept RITE_REQ do
              MODIFY:=TRUE;
           end RITE_REQ;
     or
           when MODIFY=TRUE
           = >
           accept RITE_REL do
              MODIFY:=FALSE;
           end RITE_REL;
     end select;
  end loop;
end READ_WRITE;
```

Example 5-6

component. With this approach, one can then implement "Action procedures" in the same manner outlined on page 116 of the BBN report. This implementation is as shown in Example 5-7.

Alternately, instead of using an interlock object of type RESOURCE, one could employ the interlock mechanism described previously.

5.1.3 Naming Convention Problems

5.1.3.1 Statement of Problem

The authors of the BBN Report feel that Ada's task naming conventions do not allow the programmer to name processes with names which accurately reflect the underlying algorithm structure. In particular, they feel that the array structure imposes a relatively arbitrary task structure which may or may not reflect the nature of the particular application.

A second, potentially more serious problem, is posed by the asymmetry of knowledge between the called and the calling task in which a server task has no way to reply to a requesting task outside of the rendezvous since the identity of the requesting task is not known by the server. This is called the "return address problem" by the authors. Note that the problem is not one of authenticating a requestor but rather one of being able to identify the requestor in a subsequent entry call.

5.1.3.2 Alternatives

The first problem discussed above is no longer applicable. Due to revised Ada's treatment of tasks as types, task objects may now be created and named in a meaningful manner with names more closely associated with the underlying process structure.

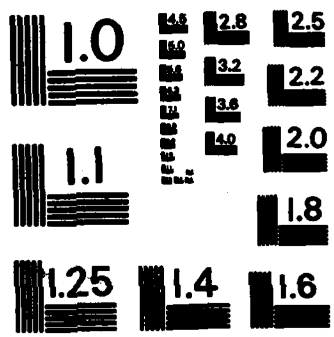
Furthermore, the limitations of the array structure of tasks in preliminary Ada are no longer present. In preliminary Ada, one could not easily build linked lists of task objects (or any other structure of task objects besides arrays, for that matter). This problem no longer exists, as

```
task type RESOURCE is
  entry SEIZE;
  entry RELEASE;
end RESOURCE;
task body RESOURCE is
  FREE:boolean:=TRUE;
begin
  loop
     select
        when FREE =>
           accept SEIZE do
              FREE:=FALSE;
           end SEIZE;
     or
           accept RELEASE do
              FREE:=TRUE;
           end RELEASE;
     or
        when FREE=>
           terminate;
     end select;
  end loop;
end RESOURCE;
     .
     .
     .
type ENTITY is
  record
     INTERLOCK:RESOURCE;
     -- other necessary declarations
  end record;
     .
     .
     .
type E_PNTR is access ENTITY;
     .
     .
     .
procedure ACTIONn(ENT:E_PNTR) is
begin
  ENT.INTERLOCK.SEIZE;
  -- perform action n
  ENT.INTERLOCK.RELEASE;
end ACTIONn;
```

Example 5-7

AD-A121 938 EVALUATION OF ADA AS A COMMUNICATIONS PROGRAMMING LANGUAGE(U) SYSCON CORP SAN DIEGO CA A L BRINTZENHOFF ET AL. 31 MAR 81 DCA100-80-C-0037 2/4

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

illustrated by Example 5-8, an implementation of a linked list of terminal drivers. Note that this example reflects the solution to the terminal driver problem cited in the BBN report.

The "return address problem" can be handled within the Ada language by defining access types which provide pointers to task types as shown in Example 5-9.

The solution to the "return address problem" is accomplished in the above example by including an ACCEPT statement within USER task's body that is used to establish the USER's own identity. The parameter passed within the ACCEPT statement of task SERVER is a pointer to the USER task itself and is saved locally. The USER task is then free to perform a call to the associated SERVER task and pass its own identity (pointer) with the call. The USER task then loops forever carrying out necessary processing while it awaits a reply from the SERVER task. The SERVER task is able to know (and remember) the identity of the USER task since it was passed as a parameter (pointer) upon ENTRY call, and then stored locally.

5.1.4 Lack of Scheduling Control

5.1.4.1 Statement of Problem

The BBN Report expresses concern as to whether the scheduling discipline provided by the language is sufficiently general to support applications with important timing constraints, and in particular, whether Ada provides adequate control over the scheduling strategy.

5.1.4.2 Alternatives

It is believed that the example provided by the authors on pages 130 and 131 of the BBN Report outlines considerations traditionally handled by an executive/operating system - not within a language definition. Requiring the Ada definition to encompass task run-time limit specification and/or forcible descheduling is above and beyond the requirements of a language definition. It should be emphasized that Ada does not provide any scheduling discipline, but rather

```
package TERM_DRVR_SYSTEM is
  task TERM_DRVR is                    -- Terminal Driver Specification
     entry START_UP(N:NATURAL);        -- Activation entry
     entry SHUT_DOWN;                  -- Deactivation entry
  end TERM_DRVR;                       -- Assumes only one user
  type TERM;
  type TERM_PNTR is access TERM;
  type TERM is
     record
        PNTR: TERM_PNTR;
        DRVR: TERM_DRVR;
        T_NUM: INTEGER range 0..INTEGER'LAST:=0;
     end record;
  FREE_TERM_DRVR: TERM_PNTR;
  ACTIVE_TERM_DRVR: TERM_PNTR;
  procedure BUILD_FREE_LIST(N:NATURAL);
  procedure ACTIVATE (ACTERM:NATURAL);
  procedure DEACTIVATE (ACTERM:NATURAL);
  STATUS_ERROR:exception;
end TERM_DRVR_SYSTEM;

package body TERM_DRVR_SYSTEM is
-- TERM_DRVR represents the terminal
-- driver task which will monitor and
-- interface to a particular terminal
-- represented by a positive integer.
-- Details of the interaction are not
-- presented here...only the capabilities
-- to start and stop the actions of the
-- terminal driver.
  MY_TERM:NATURAL;                     -- Terminal Number
  READY:BOOLEAN:=TRUE;                 -- Initialization flag
  task body TERM_DRVR is
  begin
     loop
        select
           when READY =>
              accept START_UP(N:NATURAL)do
              MY_TERM:=N;
              READY:=FALSE;
              -- perform any START_UP processing
              end START_UP;
        or
           accept SHUT_DOWN do
           -- perform any housecleaning
           READY:=TRUE;
           end SHUT_DOWN;
        end select;
     end loop;
  end TERM_DRVR;
  --
```

Example 5-8. (Page 1 of 3)

```
procedure BUILD_FREE_LIST(N:NATURAL) is
-- This procedure is used to initially
-- create a list of N FREE
-- terminal driver task objects in
-- a simple linked list structure
   TEMP:TERM_PNTR;
begin
   FREE_TERM_DRVR:=new TERM;  -- point to head
   TEMP:=FREE_TERM_DRVR;
   for I in 2..N loop         -- add N-1 nodes
      TEMP.PNTR:=new TERM;    -- to linked list
      TEMP:=TEMP.PNTR;
   end loop;
   TEMP.PNTR:=null;
end BUILD_FREE_LIST;
--
procedure ACTIVATE(ACTERM:NATURAL) is
-- This procedure is used to remove
-- terminal driver tasks from the
-- FREE list and place them on
-- an ACTIVE list.  Note that a
-- particular task is associated with
-- a terminal via a specified terminal
-- number
TEMP:TERM_PNTR;
begin
   if FREE_TERM_DRVR = null then -- test for any drivers
      raise STATUS_ERROR; -- error if none
   end if;
   TEMP:=FREE_TERM_DRVR; -- remove head from FREE
   FREE_TERM_DRVR:=FREE_TERM_DRVR.PNTR; -- terminal driver
   TEMP.T_NUM:=ACTERM; -- save terminal number
   TEMP.PNTR:=ACTIVE_TERM_DRVR; -- place at head of ACTIVE
   ACTIVE_TERM_DRVR:=TEMP;
   ACTIVE_TERM_DRVR.DRVR.START_UP(ACTERM); -- start driver
end ACTIVATE;
--
```

Example 5-8. (Page 2 of 3)

```
  procedure DEACTIVATE(ACTERM:NATURAL) is
  -- This procedure is used to remove active
  -- terminal drivers from the ACTIVE list
  -- and place them back on the FREE
  -- list.
  TEMP:TERM_PNTR;
  LAST_PNTR:TERM_PNTR;
  begin
     TEMP:=ACTIVE_TERM_DRVR;
     if TEMP.T_NUM=ACTERM then
        TEMP.DRVR.SHUTDOWN;
        ACTIVE_TERM_DRVR:=TEMP.PNTR;
        TEMP.PNTR:=FREE_TERM_DRVR;
        FREE_TERM_DRVR:=TEMP;
        return;
     end if;
     LAST_PNTR:=TEMP;
     while TEMP /= null loop
           if TEMP.T_NUM=ACTERM then
                 TEMP.DRVR.SHUT_DOWN; -- disable driver
                 LAST_PNTR.PNTR:=TEMP.PNTR; -- remove from list
                 TEMP.PNTR:=FREE_TERM_DRVR; -- place at head of
                 FREE_TERM_DRVR:=TEMP; -- FREE list
                 return;
           end if;
           LAST_PNTR:=TEMP;
           TEMP:=TEMP.PNTR;
     end loop;
     raise STATUS_ERROR; -- no driver error
  end DEACTIVATE;
end TERM_DRVR_SYSTEM;

with TERM_DRVR_SYSTEM; use TERM_DRVR_SYSTEM;
procedure MAIN is
begin
  BUILD_FREE_LIST(50); -- set up 50 node FREE list
  loop
  -- if terminal n needs a driver
  ACTIVATE(n);
  -- or if terminal n is done
  DEACTIVATE(n);
  end loop;
end MAIN;
```

Example 5-8. (Page 3 of 3)

```
procedure MAIN is
  type MESSAGE is...;   -- some form of message to process
  type ANS is...;       -- some form of server response

  type USER;
  type ACC_USER is access USER;
  task type USER is
     entry NAME(N:ACC_USER); -- used to get own name
     entry ANSWER (A:ANS);   -- used for server response
  end USER;

  type U_INFO is
     record
     USER_ID:ACC_USER;
     MSG:MESSAGE;
  end record;

  task type SERVER is
     entry CALL(U:U_INFO);
     entry SHUT_DOWN;
  end SERVER;
  SERVE:SERVER;
  task body SERVER is separate;
  task body USER is separate;

begin
  declare
     TEMP:ACC_USER;
  begin
     while WANT_TO_BUILD_USERS loop
        TEMP:=new USER;
        TEMP.NAME(TEMP);
     end loop;
  end;
end MAIN;  -- wait for users and servers to complete
```

Example 5-9 (Page 1 of 3)

```
task body SERVER is
  SIZE:=constant INTEGER:=...;          -- some max buffer size
  U_RECS:array (1..SIZE) of U_INFO;  -- user request buffer
  IN:INTEGER range (1..SIZE):=1;      -- buffer input index
  OUT:INTEGER range (1..SIZE):=1;     -- buffer output index
  COUNT:INTEGER range (0..SIZE):=0;  -- num items in buffer
  A:ANS;                            -- some form of server response
begin
  loop
     while CALL'COUNT>0 and COUNT<SIZE loop
        accept CALL(M:U_INFO) do
        U_RECS(IN):=M;
        end CALL;
        IN:=IN mod SIZE+1;
        COUNT:=COUNT+1;
     end loop;
     while CALL'COUNT=0 and COUNT>0 loop
        -- process the request
        -- for service to one user
        -- at a time.
        U_RECS(OUT).USER_ID.ANSWER(A);
        OUT:=OUT mod SIZE+1;
        COUNT:=COUNT-1;
     end loop;
     select
        when CALL'COUNT=0 and COUNT=0=>
           accept SHUT_DOWN;
           exit;
        else
           null;
        end select;
     end loop;
  end SERVER;
```

Example 5-9 (Page 2 of 3)

```
task body USER is
-- representative one of potentially many
-- user tasks that may request service
-- from single server task
INFO:U_INFO;
begin
  accept NAME(N:ACC_USER) do  -- get own name
     INFO.USER_ID:=N;
  end NAME;
  -- carry out processing to build
  -- message cr data to have
  -- processed by server
  SERVE.CALL(INFO);  -- call server task
  loop
     select
        accept ANSWER(A:ANS) do
           -- process response from server
        end ANSWER;
        exit;
     else
        -- carry out alternative processing
        -- while waiting for answer
     end select;
  end loop;
  -- other processing, as required
end USER;
```

Example 5-9 (Page 3 of 3)

provides a task interaction mechanism to be used however the user wishes.

5.2 COMMUNICATION SYSTEM RELATED CONCURRENCY ISSUES

Sections 2 and 3 laid the foundation for considering a model on which one may build a particular implementation for analysis. Section 4 established general concurrency facilities traditionally used for process control as well as those facilities supported within the Ada language and how they apply to the SCI model. This section will address deficiencies of the Ada language tasking constructs discovered by mapping the Ada facilities onto the SCI model. Alternatives or solutions to these deficiencies are also presented, where possible.

5.2.1 <u>Operating System Requirements</u>

5.2.1.1 <u>Statement of Problem</u>

A minimum operating system framework sufficient to support the Ada tasking constructs as well as the SCI model architecture would have to contain the following capabilities:

- Scheduler software
- Task context switching software
- Task activation table storage and queuing structure
- Memory allocation and mapping mechanism
- Interval timing mechanism and associated software
- Means to associate hardware interrupts with interrupt service routines and tasks
- I/O interface(s) to a general complement of peripheral equipment

The problem lies in the fact that these operating system requirements could potentially impact the smaller hardware environments that currently support communication systems.

5.2.1.2 Alternatives

If future communication system implementations were to follow the patterns of past and present implementations then the above stated problem would indeed be valid. However, communication systems design efforts, like other state-of-the-art embedded computer systems design efforts, are turning away from the general purpose processor environment (and its associated operating system) and turning towards implementations which exhibit a greater number of smaller dedicated distributed processors. In these environments more emphasis will be placed on hardware/firmware support of what were once traditional operating system tasks. A single task running on a single processor obviously doesn't require the OS support described above. Furthermore, what operating system software there is will be dedicated rather than general purpose and will almost certainly be written in the same high level language used for the application software.

The point being made is this. The above stated problem is currently valid. However, as times goes on, it becomes less of a problem since future design directions will eventually minimize the impact. The time frame in which this will happen should conveniently coincide with the introduction of Ada compilers into the user domain.

5.2.2 Scheduling Deficiencies

5.2.2.1 Statement of Problem

There are actually two issues included in this category. One of these issues coincides with the previously stated BBN issues. It will again be discussed here, however, for completeness.

First, it should be pointed out that certain assumptions are made as to the implementation of the scheduling algorithm. For the sake of simplicity, the criteria given in the Ada Rationale are used /ICHB79b/. These are as follows:

- The processor is available
- A new task is placed on the ready queue of the scheduler

- The scheduler is an external process (operating system)
- Ready queue is examined top to bottom and the first task ready to execute will be invoked

It should also be noted that under the Ada rendezvous concept, the scheduler will be invoked according to the following events:

- Initiation of a task
- Termination of a running task
- Entry call
- Reaching an accept statement for which no call has been issued or a select statement for which there is no possible alternative for immediate execution
- Termination of a rendezvous
- Execution of a delay statement
- Expiration of a delay
- Reception of an interrupt awaited by a task

The first problem encountered closely resembles the first listed BBN issue. The problem is that the rendezvous concept associates interprocess communication with synchronization in time in all cases. In a communication environment synchronization is not always required or desired. The inability to optionally specify whether synchronization (rendezvous) is to take place during interprocess communication is the deficiency.

The second problem deals with the inability to directly manipulate queues within the available Ada framework. Ada has chosen a FIFO implementation for task queuing at the expense of all others.

5.2.2.2 Alternatives

The fact that Ada requires a calling process to synchronize in time with a called process in order to directly perform interprocess communication is an unfortunate side effect of the rendezvous mechanism. The problem here is not so much one of inefficiency or scheduler delays, but rather one of

communication system requirements in which interprocess communication is desired while synchronization is not. The message passing alternative presented in Section 5.1.1.2 again becomes a viable mechanism in these situations. Another obvious alternative is to provide an intermediate buffering task which is dedicated to receiving and sending messages between application tasks. This allows a sending process to deposit a message with the buffering task and proceed with its appointed tasks without waiting for a receiving task to rendezvous, as shown in Example 5-10. Note that message context switching is avoided by the use of access types.

The problem of dynamic queue manipulation is not directly addressed by the available language constructs. The FIFO nature of the entry queue might be thought to be a severe constraint in cases where some requests may be of high priority. The handling of requests with priorities is achieved by the use of separate entries for each level. As shown in the Ada Rationale, a family can be conveniently used for this purpose. See Example 5-11.

Note that this approach only addresses a small number of priority levels. Efficient handling of large numbers of requests with priorities in a realistic, flexible manner is possible but beyond the scope of this report.

5.2.3 Mutual Exclusion

5.2.3.1 Statement of Problem

This problem area addresses the inflexibility of the Ada tasking constructs in much the same light presented within the BBN Report. It has been shown that mutual exclusion is a necessary aspect of parallelism within communication systems. The only vehicle for mutual exclusion directly available within the Ada language is the task rendezvous. However, this construct is inefficient in situations which only require mutually exclusive access to shared objects and which are not concerned with synchronization and/or interprocess communication.

```
package TGT_SYS is
-- TGT_SYS describes the characteristics
-- of tasks which can receive messages
-- asynchronously from a sender task
  type MSG IS ...; -- some form of message
  task type TGT_TASK is
     entry MSG_RCVR(M:MSG); -- entry to receive msgs
  end TGT_TASK;
  type ACC_TGT is access TGT_TASK; -- access value used as an
  procedure SEND_MSG(T:ACC_TGT;M:MSG); -- msg delivery addr
end TGT_SYS;

package body TGT_SYS is
  task type MSG_CARRIER is -- acts as mailman
     entry TEXT(T:ACC_TGT;M:MSG);
  end MSG_CARRIER;

  type ACC_MSG is access MSG_CARRIER;

  task body MSG_CARRIER is
  -- accepts a message and
  -- to whom to deliver it.
  -- Then attempts delivery...
  -- will terminate
  -- following delivery
     T1: ACC_TGT;
     M1: MSG;
  begin
     accept TEXT(T:ACC_TGT;M:MSG) do
        T1:= T;
        M1:= M;
     end;
     T1.MSG_RCVR(M1);
  end MSG_CARRIER;

  procedure SEND_MSG(T:ACC_TGT;M:MSG) is
  -- will dynamically create
  -- mailman tasks in a
  -- uniform manner...
  -- existence of mailman
  -- depends on access type
  -- not this procedure
     TEMP: ACC_MSG:= new MSG_CARRIER;
  begin
     TEMP.TEXT(T,M);
  end SEND_MSG;
```

Example 5-10 (Page 1 of 2)

```
  task body TGT_TASK is
     .
     . -- accept MSG_RCVR
     .
  end TGT_TASK;
end TGT_SYS;

with TGT_SYS; use TGT_SYS;
procedure MAIN is
  task type USER is -- sender of messages
     .
     .
     .
  end USER;
  type ACC_USER is access USER;
  TGT_ARRAY: array(1..n) of ACC_TGT;
  USER_ARRAY: array(1..m) of ACC_USER;
  task body USER is
  -- USER sends messages to
  -- TGT_TASKS in an
  -- asynchronous manner
     U_MSG: MSG;
     J: INTEGER range TGT_ARRAY'RANGE;
  begin
     -- create message in U_MSG;
     -- set J to index of target task in TGT_ARRAY
     SEND_MSG(TGT_ARRAY(J),U_MSG);
     -- continue processing
  end USER;
begin
  for I in TGT_ARRAY'RANGE loop
     TGT_ARRAY(I):= new TGT_TASK;     -- create target tasks
  end loop;
  for I in USER_ARRAY'RANGE loop
     USER_ARRAY(I):= new USER;        -- create user tasks
  end loop;
end MAIN;
```

Example 5-10 (Page 2 of 2)

```
task CONTROL is
  type LEVEL is (URGENT,MEDIUM,LOW);
  entry REQUEST (LEVEL'FIRST..LEVEL'LAST)(D:DATA);
end;
task body CONTROL is
  loop
     select
        accept REQUEST(URGENT)(D:DATA) do
        -- high priority processing
        end;
     or when (REQUEST(URGENT)'COUNT=0)=
        accept REQUEST(MEDIUM)(D:DATA) do
        -- medium priority processing
        end;
     or when((REQUEST(URGENT)'COUNT=0) and
             (REQUEST(MEDIUM)'COUNT=0))=
        accept REQUEST(LOW)(D:DATA) do
        -- low priority processing
        end;
     end select;
  end loop;
end CONTROL;
```

Example 5-11

5.2.3.2 Alternatives

As previously stated, the intent of the Ada design team was to incorporate one mechanism into the language which could address all three aspects of concurrent process control: process synchronization, process communication, and mutual exclusion. The requirements for simplicity of use, reliability and maintainability were seen as taking precedence over efficiency of implementation. Again, if more efficient primitive mechanisms are desired simply to provide mutually exclusive access to shared objects within a critical region, they can be implemented. Section 5.1.2.2 provided a representative sampling of candidate mechanisms.

5.2.4 Dynamic Task Priority Assignment

5.2.4.1 Statement of Problem

The ability to dynamically change task priorities is a desirable feature to have when dealing with momentary, heavy resource load or casualty conditions. It is also a convenient method of handling the so-called "starvation effect" whereby a low-priority task never gets scheduled due to continual preemption by higher priority tasks.

Preliminary Ada, as documented in /ICHB79a/, provides for a dynamic or static assignment (pragma) of task priority. The current documentation /USDO80/ has dropped the dynamic flexibility.

5.2.4.2 Alternatives

As in so many previous cases, the decision to remove dynamic task priority manipulation from the language was based on a conscious decision on the part of the design team. Again, reliability and maintainability of generated code took precedence over the convenience of including this capability within the language. And, as before, the above mentioned feature can be implemented with available constructs and data structures; however, the solutions are not as direct.

One indirect method of solving this problem is to use duplicate "instances" of tasks each having a different statically assigned priority. As exceptional conditions occur, the appropriate priority task object is "spawned" using access pointers. When the exceptional conditions cease to exist, the task objects may be deallocated and the allocated space may be reclaimed with an available "garbage collection" mechanism. Obviously, what priority scheme is used and which mechanisms are employed to reclaim deallocated space are considerations which will be functions of the particular application.

One example of this method of handling priorities might be to spawn a duplicate copy of an executing task (at a higher priority) from an exception handler buried within the executing task. This has the effect of artificially raising one's own priority. Another example might be to define duplicate copies of diagnostic routines at each priority level (assume there are three: low, medium, and high). Under normal conditions a channel diagnostic, for example, may be requested to isolate faults on a particular channel on a background (low priority) basis. This may result from an operator action at a monitoring console, for example, and would cause the system management routine to spawn this low priority copy of the channel diagnostic task. In a casualty situation, an executing application task might raise an exception in response to detection of a fault on a message transmission and signal the system manager to immediately spawn a high priority channel diagnostic task, perhaps preempting other executing application tasks.

A possible third method might employ dedicated server tasks at predefined priority levels whose only purpose is to receive requests from application tasks to spawn a desired diagnostic task. The rendezvous associated with the application task/server task linkage will be executed at the higher priority of the two tasks. Similarly, the server task/diagnostic task rendezvous will be executed at the higher priority of these two tasks. Thus, if an application task calls the highest priority server task which subsequently calls

the appropriate diagnostic task, the nature of the priority mechanism dictates that, at least during rendezvous, all statements within the body of the accept statement will be executed at the higher priority. Therefore, one could place all desired high priority statements within the context of the end task's accept statement to guarantee high priority execution. Obviously, there are many variations of these examples and the implementation of the particular method will be dependent upon the application in question.

5.3 MISCELLANEOUS ISSUES

The main emphasis of the analysis effort was to examine Ada concurrency features and how they can be applied to communication system requirements. In the course of this analysis another non-concurrency related issue surfaced.

5.3.1 Dynamic Record Structure Manipulation

5.3.1.1 Statement of Problem

Generally, a communication system architecture is layered according to functional specification. Moreover, the type of processing that occurs on data structures (message buffers, packets, and headers) varies from layer to layer within a given architecture. Usually, the upper layers will generate headers and manipulate user data at the character, string, or array levels, while the lower layers will view the same data at a bit level. This requires the ability to dynamically represent and access particular data structures in different manners at different points during execution. Ada does not provide a convenient direct way to perform this manipulation. In Ada, the set of values of a record type discriminant must be statically determined at compile time. In order to change values (or form) at run time, it is necessary to perform a complete record assignment which could be extremely cumbersome.

5.3.1.2 Alternatives

Run time structure manipulation was specifically prevented for reliability reasons. However, there is a potential means of performing dynamic record structure manipulation if one so desires. Example 5-12 shows how one may employ the generic function UNCHECKED_CONVERSION to dynamically convert record structures. If the records in question are large, this alternative may result in inefficient data context swapping. Therefore, a further alternative is provided by Example 5-13, which shows an implementation employing unchecked conversion on the pointers to the records.

```
with UNCHECKED_CONVERSION;
      .
      .
      .
type FORMATTED_MESSAGE is
  record
     ID:INTEGER range 0..255;
     D:DATE;
     M:STRING(1..10);
  end record;
type UNFORMATTED_MESSAGE is
  array (1..FORMATTED_MESSAGE'SIZE) of BOOLEAN;
pragma PACK(UNFORMATTED_MESSAGE);
function DECODE is new
  UNCHECKED_CONVERSION(FORMATTED_MESSAGE, UNFORMATTED_MESSAGE);
function ENCODE is new
  UNCHECKED_CONVERSION(UNFORMATTED_MESSAGE, FORMATTED_MESSAGE);
      .
      .
      .
MSG:FORMATTED_MESSAGE:= (5,(8,OCT,1947),"HI THERE  ");
BITS:UNFORMATTED_MESSAGE;
      .
      .
      .
BITS:=DECODE(MSG);
MSG:=ENCODE(BITS);
```

Example 5-12

```
with UNCHECKED_CONVERSION;
      .
      .
      .
type FORMATTED_MESSAGE is
  record
  ID:INTEGER range 0..255;
  D:DATE;
  M:STRING(1..10);
  end record;
type ACCESS_FM is access FORMATTED_MESSAGE;
type UNFORMATTED_MESSAGE is
  array (1..FORMATTED_MESSAGE'SIZE) of BOOLEAN;
pragma PACK(UNFORMATTED_MESSAGE);
type ACCESS_UM is access UNFORMATTED_MESSAGE;
MSG:ACCESS_FM:=new FORMATTED_MESSAGE;
BITS:ACCESS_UM:=new UNFORMATTED_MESSAGE;
function DECODE is new
  UNCHECKED_CONVERSION(ACCESS_FM,ACCESS_UM);
function ENCODE is new
  UNCHECKED_CONVERSION(ACCESS_UM,ACCESS_FM);
      .
      .
      .
-- assume message transmission results
-- in buffer arriving in binary format
-- in object of UNFORMATTED_MESSAGE type.
-- By performing ENCODE operation on access
-- types, you can now access fields of
-- FORMATTED_MESSAGE objects.
      .
      .
      .
MSG:=ENCODE(BITS);
if MSG.ID = 4 then
  MSG_TYPE_FOUR_PROCESSOR;
end if;
      .
      .
      .
```

Example 5-13

(This Page Intentionally Left Blank)

SECTION 6
EVALUATION OF PROPOSED ALTERNATIVES

This section presents the details of an overall evaluation of those alternatives which were proposed in Section 5. The evaluation will be performed from the viewpoint of (1) the efficiency of implementation of the alternatives and (2) the effectiveness of the alternatives themselves.

6.1 DEFINITION OF CRITERIA

6.1.1 Efficiency Criteria

Efficiency can be defined as a measure of the ability to do a job versus the cost incurred. Specifically in terms of software, it can be defined as a measure of the amount of computing resources and code required by a program to perform a particular function. Efficiency only can truly be measured, and hence realistically evaluated by empirical observation of the software operating in a controlled test environment. Since no compiler is currently available by which empirical data may be obtained, it is necessary to resort to a somewhat subjective evaluation. However, when a legitimate compiler becomes available, then exhaustive test and evaluation of both the built-in and constructed Ada process control mechanisms previously proposed can be performed.

In the meantime, the proposed alternatives will be evaluated on the basis of two efficiency criteria: execution efficiency and storage efficiency. These criteria are defined in the following manner:

- Execution Efficiency - a measure of the ability of the alternative to provide for minimum processing time.
- Storage Efficiency - a measure of the ability of the alternative to provide for minimum storage requirements during operation.

6.1.2 Effectiveness Criteria

Effectiveness can be defined as a measure of how well something does a job for which it was designed. In particular, the effectiveness of software can be defined as a measure of the extent to which a program satisfies its requirements and fulfills its intended functional and operational objectives.

Again, to properly measure the effectiveness of a particular mechanism or proposed alternative, one needs to gather empirical data. However, some qualitative assessment of the effectiveness of the proposed alternatives may be made on the basis of the following definitions of various criteria:

- Usability - a measure of how easily an alternative may be applied to the problem at hand, i.e., ease of programmer specification.
- Manageability - a measure of how easily one can control the alternative in use.
- Reliability - a measure of how accurately an alternative repeatedly performs its intended function.
- Documentability - a measure of the ability of an alternative to be self-documenting.
- Portability - a measure of the ease by which an alternative may be applied to a similar but distinct problem.
- Maintainability - a measure of the ability of an alternative to withstand changes - to itself or to its environment.

6.2 EVALUATION OF ALTERNATIVES

6.2.1 Evaluation of Alternatives to BBN Report Criticisms

This section presents evaluations of the efficiency and effectiveness of the various alternatives to the BBN criticisms detailed in Section 5.

6.2.1.1 <u>Excessive Scheduler Interactions</u>

The buffered message passing example (Example 5-1) offers an efficient alternative to the problem of excessive scheduler interactions associated with a strict task rendezvous implementation. In fact, it was shown that the rendezvous mechanism for buffering operations was avoided through use of a shared queue with an associate interlock. The execution efficiency of this alternative is totally dependent on the nature of the application. Since a spin lock mechanism is used for those tasks awaiting access to the queue, any situation which results in inordinate amounts of busy waiting will certainly undermine the efficiency of the alternative. In fact, if system performance analysis indicates that the busy waiting time approaches the overhead associated with scheduler interactions, the conventional Ada rendezvous mechanism would be more appropriate. However, the assumption here is that on the average this busy wait time is small compared to scheduler overhead.

In examining the effectiveness, one can see that this alternative provides an effective means of avoiding the scheduler delays associated with a strict Ada rendezvous. The method employed follows more traditional bounded buffer manipulation methods and is hence easy to use and manage. Reliability is not really a concern since the code is simple and straightforward and does not lend itself to errors. The method is conceptually portable in that it may be employed in any situation requiring buffered message passing with non-blocking senders. The disadvantage lies in the utilization of the interlock mechanism whereby deadlock can occur in any situation leading to unmatched pairs of LOCK and UNLOCK operations. Thus, maintenance becomes a concern since the compiler does not enforce this "synchronization" as it does in the case of the Ada rendezvous.

6.2.1.2 <u>Process Control Structure Inflexibility</u>

This area actually involves two separate problem areas. The first concern is related to inflexibility of Ada's process control mechanisms. The second one involves the problem of the synchronization mechanism being control-based instead of data-based. The evaluation of the alternatives to each of these problem areas will be presented separately below.

6.2.1.2.1 <u>Process Control Mechanism Inflexibility</u>

As an answer to this problem area, Section 5 presented four alternatives ranging from low level (interlock) mechanisms to high level (monitor) mechanisms. Each of the examples offers an alternative to the use of the direct entry/accept linkage for mutual exclusion.

Considering the efficiency of the proposed alternatives, the interlock implementation using assembly language offers the most efficient mechanism in terms of execution time and space. These mechanisms, however, tend to exhibit the same disadvantages and advantages described in Section 4 for interlocks in general. That is, their effectiveness is limited by the fact that they are difficult to manage (control) and furthermore tend to make the code in which they occur difficult to maintain. As mentioned previously, the problem is one of "enforcing" implementation of matched pairs of "LOCK" and "UNLOCK" operations.

The binary and integer semaphore examples exhibit roughly the same characteristics. They are somewhat less efficient than the interlock mechanisms but are easier to control since the nature of the entry/accept linkage of the P and V operations forces a sequential ordering of the calls. However, like the interlock, they can be abused. As noted in the Ada Rationale /ADARAT79/, they exhibit problems which severely limit their effectiveness. The advantages are their relative efficiency, ease of programmer specification, and documentability. The integer semaphore may be viewed as simply a more flexible implementation of the binary semaphore. The advantages and disadvantages may be similarly applied.

The last example illustrates an implementation of a monitor in Ada. Advantages lost in efficiency of execution and space are gained in effectiveness of implementation. The monitor implementation is probably the least efficient of any of the process control mechanism Ada implementations. It also tends to be more difficult to implement and use. However, it is reliable, manageable (once implemented) and lends itself very well to maintenance since all operations and protected data are centralized within the monitor itself.

The conclusion reached in the evaluation of these four alternatives is that they are mechanisms which offer various trade-offs in advantages and disadvantages and a decision as to which one to apply to a particular situation should be based on the requirements of the situation. The point is that Ada does provide the implementor with the capability to construct a wide range of process control mechanisms with which to work.

6.2.1.2.2 Storage Inefficiency of Control-Based Synchronization Mechanisms

The problem of storage inefficiency associated with Ada's control-based synchronization mechanism was addressed by Example 5-7. In this example, the solution was to define pointers to each of the entities which are defined as records containing interlock objects as their components.

It can be seen that this produces an efficient solution in terms of storage since the objects are created on an as-needed basis using access types. Note that the execution efficiency can be improved by employing the previously described interlock instead of the interlock of type RESOURCE.

The solution offered is a very straightforward implementation, though the usefulness and manageability is dependent on the availability and controlled use of some garbage collection mechanism. This is an assumption which also governs the portability of the solution. Finally, the solution is thought to be very readable and easily maintained.

6.2.1.3 Naming Convention Problems

Again, there are actually two distinct problem areas in this category. The first problem area is that Ada's task naming conventions do not allow the programmer to name or create tasks which accurately reflect the underlying algorithm structure. In other words, the array structure of task families did not allow one to create task objects with meaningful names. Moreover, the array structure did not easily map onto any underlying structure except arrays. The second, potentially more serious problem is posed by the asymmetry of knowledge between the called and the calling task.

6.2.1.3.1 Task Naming/Structure Inconsistency

The problem of not being able to meaningfully name tasks is no longer applicable. Tasks are now defined as types and named objects may be created to meaningfully match the underlying structure.

Furthermore, the limitations of the array structure of tasks in preliminary Ada are no longer present. Example 5-8 illustrates an implementation of a linked list of terminal drivers which addresses the terminal driver problem cited in the BBN report. Obviously, this is a much more efficient implementation than the three preliminary Ada alternatives listed on page 123 of the BBN report.

In terms of effectiveness, it can be seen that the alternative presents the most direct solution to the stated problem. In fact, it satisfactorily meets all of the defined effectiveness criteria.

6.2.1.3.2 Return Address Problem

There is no direct mechanism available in Ada whereby a server task can know the identity of its customers. Example 5-9 provides an indirect method to solve this problem. In the example, an accept statement within the customer task's body is used to establish the customer's own identity, which is then passed as a parameter (using a pointer) to the server task.

Because the mechanism employs pointers to the customers, it is considered to be a fairly efficient solution in terms of both execution time and storage. Because it is an indirect mechanism, it is seen to be less effective than a mechanism which could be built into the language to provide symmetry of knowledge between customer and server tasks. This is because it is somewhat difficult to use, manage, and maintain. For example, if the programmer neglects giving the customer task its own task name, the whole scheme breaks down.

6.2.1.4 Lack of Scheduling Control

Not applicable for reasons cited in Section 5, paragraph 5.1.4.2.

6.2.2 Evaluation of Alternatives to Other Communication-Related Concurrency Issues

This subsection presents evaluations of the efficiency and effectiveness of the various alternatives to other communication-related concurrency issues outlined in Section 5.2.

6.2.2.1 Operating System Requirements

Not applicable for reasons cited in Section 5.2.1.2.

6.2.2.2 Scheduling Deficiencies

There are actually two issues included in this category. The first problem closely resembles the first listed BBN issue and involves the fact that Ada always associates interprocess communication with synchronization in time. The second problem deals with the inability to directly manipulate queues within the available Ada framework since Ada has chosen a FIFO implementation for task queuing at the expense of all others.

6.2.2.2.1 Interprocess Communication Problems

The alternative to this problem area was presented in Example 5-10. It involves an intermediate buffering task which is dedicated to receiving and sending messages between application tasks. This allows a sending process to deposit a message with the buffering task and proceed with its appointed tasks.

The main advantage of this alternative is that the application (user) tasks are not held up waiting for rendezvous to occur. It is not necessarily efficient in terms of overall execution time since several additional scheduler interactions may be required. In fact, in the case where the target task (receiver) is almost always in a position to rendezvous, the proposed alternative would be much less efficient in the long run. In addition, it is not necessarily storage efficient since extra storage for the intermediate task is required.

The advantage of the alternative is that it is a direct, effective means of handling situations in which application tasks cannot be delayed waiting for rendezvous to occur. It is not very easily implemented and does not lend itself to readability. It is, however, somewhat easy to control since the users must explicitly indicate the target system in question. Furthermore, it is conceptually portable and easily maintained.

6.2.2.2.2 Inability To Directly Manipulate Entry Queues

Ada does not provide the direct capability to dynamically manipulate queues of calling tasks waiting to rendezvous with a called task. The Ada Rationale provided Example 5-11 as an alternative to this problem. Even though this is a viable mechanism to solve the problem, it is unfortunately not very efficient. Note also that the example addresses only a small number of priority levels. Thus, a more sophisticated and hence less efficient mechanism would have to be employed to handle a larger number of priority levels.

The example given, however, is easily implemented, easy to control, reliable, very readable, conceptually portable, and easily maintained. As such, it is considered to be an effective solution to the stated problem.

6.2.2.3 Inefficiency of Rendezvous for Mutual Exclusion

This problem was previously addressed in paragraph 6.2.1.2.1.

6.2.2.4 Dynamic Task Priority Assignment

The ability to dynamically change task priorities is a desirable feature to have available when dealing with momentary heavy resource load or casualty conditions. Even though there is no direct mechanism available within Ada to handle this problem, paragraph 5.2.4.2 described some viable alternatives.

The alternatives described are not very efficient. In the first example, storage efficiency is poor since duplicate copies of tasks have to be maintained. Also, the spawning and subsequent execution of the duplicate tasks leads to execution time inefficiency since it will almost certainly involve scheduler interactions. The last example offered is more efficient in terms of storage since the prioritized server tasks are small dedicated tasks which only call the desired diagnostic tasks. However, execution efficiency could be adversely affected in situations where the amount of processing placed within the context of the accept statement might be excessive.

Unfortunately, these are not very effective mechanisms to employ either. They are difficult mechanisms for a programmer to specify and even more difficult to manage once implemented. Reliability is a question since it is difficult to track one's location when a fault occurs. They are somewhat readable in the sense that the task definitions offer visible evidence of the intended task priorities. As such, they are also somewhat easily maintained since the different priority tasks can be localized.

6.2.3 Miscellaneous Issues

This subsection presents an evaluation of the alternative to the one significant non-concurrency related issue that surfaced during the course of the analysis effort.

6.2.3.1 Dynamic Record Structure Manipulation

Since Ada places restrictions on dynamic manipulation of the form and contents of a record structure during runtime, it is necessary to formulate an alternative mechanism to do this. Obviously, it is desirable to be able to represent and access a particular data structure in both a formatted and an unformatted manner.

Example 5-12 presented one such method using the generic function UNCHECKED_CONVERSION to dynamically convert a record structure. The alternative presented is not very efficient in either time or space since the generic functions must be instantiated and a complete context switch of the message most likely occurs upon the conversion. Example 5-13 offers a second method employing unchecked conversion of pointers to the records, rather than the records themselves. This is obviously more efficient since the conversion is performed on the pointer, avoiding the message context swap of the previous example.

Both methods offer fairly effective means of handling the problem. They satisfactorily meet all of the defined effectiveness criteria with the exception that they are somewhat cumbersome to implement.

SECTION 7
CONCLUSIONS

7.1 SUMMARY OF ANALYSIS

Sections 2 and 3 provided the framework for the definition of requirements associated with concurrent programming in communication systems applications. The general requirements were analyzed as well as those features required by a programming language to satisfactorily implement those requirements. Section 4 then addressed traditional solutions to process control and described the means whereby the Ada programming language addresses the three aspects of concurrent programming, i.e., interprocess synchronization, communication, and mutual exclusion. Section 5 discussed alternatives to all identified problem areas. First, the issues uncovered by the BBN Report /BBNE79/ were analyzed and alternatives to these problem areas were presented. Second, problems uncovered during the analysis of communications systems requirements for concurrent programming were presented and alternatives to these problems were offered. Last, the non-concurrency related problem of dynamic record manipulation was addressed. In Section 6, definitions of efficiency and effectiveness criteria were presented, followed by a qualitative evaluation of each of the solutions to the identified problems.

7.2 CONCLUSIONS

There were six distinct criticisms listed in the BBN report, five concurrency related problems, and one non-concurrency issue. Out of all of these, only two problem areas were left unanswered. These are (1) Ada's lack of control over the scheduling discipline and (2) the operating system requirements of an Ada-based communication implementation. In fact, these may not be problem areas in some implementations for reasons given in Section 5.

One overriding observation can be made following this analysis. As a high level programming language, Ada provides the implementor with the flexibility to construct alternatives to known deficiencies. In fact, the alternatives presented in this report are merely representative samples of a wider range of potential alternatives to the identified problems. The choice of a particular alternative to a particular problem area will be governed by a determination of the efficiency and effectiveness of the alternative in question. This determination can be properly made only when the applicable environment is identified and quantitative measurements can be made. The existence of a legitimate compiler and a viable support environment are obviously necessary requirements. To the extent possible, some preliminary measures of the efficiency and effectiveness of the various alternatives can be made during Phase II of this ongoing effort. While this effort will only have access to the NYU Ada/ED Translator/Interpreter, some comparative analysis of the efficiency of the various alternatives can be performed as well as a preliminary evaluation of the effectiveness of the proposed solutions. To this end, tests should be devised to specifically address the cited problems and alternatives.

This report has served to document the evaluation of using the Ada programming language for concurrent communication system programming applications. It has addressed previously cited criticisms as well as ones discovered during the course of the analysis. As a result of this preliminary analysis, it can be concluded that the current Ada language definition can be effectively applied to communication systems applications. A quantitative proof of this conclusion will be required during follow-on efforts as the applications are identified and the necessary tools become available.

APPENDIX A
REFERENCE DOCUMENTATION

/AUTO78/ AUTODIN II Mode/VI (ADCCP) Line Control Procedures Functional Specification, Final Draft. June 1978.

/BBNE76/ Bolt, Beranek, and Newman, Development of a Communications Oriented Language, Parts 1 and 2. March 1976.

/BBNE79/ Bolt, Beranek, and Newman, The Impact of Multiprocessor Technology on High Level Language Design, Final Report. DCA Contract No. 100-78-C-0028, September 1979.

/BBNE80/ Bolt, Beranek, and Newman, Formal Specification of the Transport and Session Protocols, Draft Report. Contract No. NB79SBCA0092, 1980.

/BOCH80/ Bochman, Sunshine, Communications Protocol Design. IEEE Trans., Vol. COM-28, No. 4, April 1980.

/BOUT79/ Boute, R. T., Ada and CHILL: A joint language evaluation. Report RTB-7908. Bell Telephone Manufacturing Company, Antwerpen, August 1978.

/BRIN73/ Brinch Hansen, P., Operating System Principles. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

/BRIN78/ Brinch Hansen, P., Distributed Processes: A concurrent programming concept. Comm. ACM Volume 21, Number 11, November 1978, 934-941.

/CLAR80/ Clark, Proceedings from the Seminar on DOD Data Communications: Host-to-Host Protocol Standards. NBS, November 1980.

/CONW63/ Conway, Design of Separable Transition Diagram Compiler. Comm. ACM 6,7, July 1963.

/DCAC78/ DCA, AUTODIN II Design Executive Summary. DCA Contract No. 200-C-637, May 1978.

/DECO80/ DEC, INTEL, XEROX, The Ethernet: A Local Area Network Data Link Layer and Physical Layer Specification; Version 1.0. 1980.

/HOAR78/ Hoare, C. A. R., Communicating Sequential Processes. ACM 21, 8 (August 1978), 666-677.

/HOLT78/ Holt, R. C., et al., Structured Concurrent Programming with Operating Systems Applications. Addison-Wesley, 1978.

/ICHB79a/ Ichbiah, J. D., et al., Preliminary Ada Reference Manual. ACM SIGPLAN Notices 14, 6 (June 1979), Part A.

/ICHB79b/ Ichbiah, J. D., et al., Rationale for the Design of the Ada Programming Language. ACM SIGPLAN Notices 14, 6 (June 1979), Part B.

/JONE80/ Jones, Schwarz, Experience Using Multiprocessor Systems - A Status Report. ACM Computing Surveys, Vol. 12, No. 2. June 1980.

/KOBA78/ Kobayashi, Systems Programming Series: Modeling and Analysis. 1978.

/OSIN79/ OSI/TC97/SC16, Reference Model for Open Systems Interconnection. June 1979.

/POST80/ Postel, Internetwork Protocol Approaches. IEEE Trans., Vol. COM-28, No. 4, April 1980.

/TOBA80/ Tobagi, Multi-Access Protocols in Packet Communications Systems. IEEE Trans., Vol. COM-28, No. 4, April 1980.

/USDO80/ U.S. Department of Defense, Reference Manual for the Ada Programming Language. July 1980.

/ZIMM80/ Zimmerman, OSI Reference Model. IEEE Trans., Vol. COM-28, No. 4, April 1980.

(This Page Intentionally Left Blank)

Comparative Analysis
of the
Ada and CHILL
Programming Languages

(This Page Intentionally Left Blank)

Comparative Analysis
of the
Ada and CHILL
Programming Languages

Abstract

With the increasing use of Stored Program Control telephone exchanges, the development and use of proper software tools takes on added importance. The CCITT High Level Language (CHILL) is being developed specifically for programming of SPC exchange applications. Ada is being developed to serve as a programming standard for embedded military computer systems. In many instances the functional requirements of these two application areas coincide and as such this report examines the feasibility of Ada being used as a direct substitute for CHILL, both in the context of CHILL being a programming language, and in the context of CHILL being part of a programming environment containing CHILL, SDL, and MML. The report concludes that Ada is indeed a suitable replacement for CHILL in both contexts.

(This Page Intentionally Left Blank)

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

LIST OF ILLUSTRATIONS

(This Page Intentionally Left Blank)

EXECUTIVE SUMMARY

This report presents the results of a comparative analysis between the CHILL and Ada programming languages. The approach taken in this analysis effort was to perform an exhaustive feature-by-feature comparison of the languages and the programming environments to determine if Ada can be used as a suitable replacement for CHILL. The objective of this effort was twofold:

1) To demonstrate the suitability of Ada as a replacement for CHILL in a programming language context.
2) To demonstrate the ability of Ada to replace CHILL in a programming environment containing CHILL, SDL, and MML.

The feature-by-feature comparison presented in Section 3 demonstrates that the language differences are minor. When viewed from a circuit switching application point of view, no evidence can be found that CHILL exhibits any linguistic or functional advantage over Ada. In fact, no feature exists in CHILL that dictates choosing CHILL over Ada for any telecommunication application. Since the language feature evaluation uncovered no major differences, it is concluded that Ada is, indeed, a suitable replacement for CHILL from a programming language point of view.

The examination of the Ada Programming Support Environment (APSE) and the programming environment of CHILL, SDL, and MML is presented in Section 4. It is seen that no dependency exists between the CHILL, SDL, and MML elements, and as such, no restriction is placed on their portability because of a dependency. Furthermore, the APSE is shown to be able to support the inclusion of external program tools within its outermost level. It is concluded that Ada can replace CHILL in a CHILL/SDL/MML environment, but that a more attractive approach is to incorporate the SDL and MML tools into the APSE.

These conclusions, along with other relevant issues, are presented in Section 5. Additionally, three other reports are discussed which treat the same subject and arrive at the same basic conclusions.

In summary, it is felt that the comparative analysis described in this report has convincingly demonstrated that Ada can, in fact, be used as a suitable replacement for CHILL.

SECTION 1
INTRODUCTION

1.0 INTRODUCTION

1.1 PURPOSE

The purpose of this report is to describe the results of a comparative analysis of the CCITT High Level Language (CHILL) and the Ada programming language. This analysis, as detailed herein, was conducted by Systems Consultants, Inc. (SCI) in support of the Defense Communication Agency (DCA) under Contract Number DCA 100-80-C-0037.

1.2 SCOPE

This report presents material which provides answers to the following two questions:

1) Can Ada be used as a direct substitute for CHILL in the context of CHILL being a programming language designed for circuit switching applications?
2) Can Ada be used as a direct substitute for CHILL in the context of CHILL being part of a programming environment containing CHILL, SDL, and MML?

To answer the first question a feature-by-feature comparison will be presented. The intent of this comparison is to examine the form and function of the two languages to determine how similar they are in terms of the definition and availability of their respective features. The integration of the features within each language will be addressed and it will be shown that CHILL does not hold a distinct linguistic or functional advantage over Ada. Since CHILL was, in fact, designed for circuit switching applications, it will be shown that the answer to the first question is affirmative.

To answer the second question, the CHILL/SDL/MML environment will be examined to determine the specific relationship that exists between CHILL/SDL and CHILL/MML. Additionally, the Ada Programming Support Environment (APSE) will be examined to determine its ability to support external tools such as SDL and MML. It will be shown that CHILL, SDL, and MML are not dependent on each other, that the APSE can support the incorporation of SDL/MML, and that incorporation of SDL and MML into the APSE represents an attractive formulation of a programming environment for circuit switching applications.

The report is organized in the following manner. The following section, Section 2, presents a high level overview of Ada and CHILL. This overview will include a brief history of their respective development efforts, a description of the language design goals, and current development status. This is mainly intended to provide the uninitiated reader with pertinent background information.

Section 3 presents the feature by feature comparison organized into the following subsections for convenience:

- Lexical Elements
- Data Typing
- Names, Expressions, and Statements
- Program Structure
- Concurrency
- Exception Handling
- Input/Output

Differences between the form and function of Ada's features and those of CHILL will be detailed. It will be shown that Ada and CHILL are technically very similar in terms of the definition and availability of their respective features.

Section 4 addresses the issue of a programming support environment. The Ada Programming Support Environment (APSE) will be examined to determine its interaction, compatibility, and implementation requirements. The programming environment of CHILL, SDL, and MML will be examined in similar fashion. The method and feasibility of replacing

CHILL with Ada will then be evaluated. It will be shown that an environment consisting of APSE hosting SDL/MML is the most practical and that in this context, Ada can replace CHILL.

Section 5 presents the overall conclusions of the Ada/CHILL comparative analysis effort.

(This Page Intentionally Left Blank)

SECTION 2

ADA/CHILL OVERVIEW

2.0 OVERVIEW OF ADA AND CHILL

Prior to conducting an in-depth comparison of Ada and CHILL, it is advantageous to present a brief high level overview of the languages. It will be seen that, at least superficially, the stated goals, development histories, and overall features of the languages are not at all dissimilar.

2.1 ADA

The Ada programming language is being developed by the Department of Defense (DoD) to serve as a programming standard for embedded military computer applications; e.g., shipboard, communications, avionics, or command and control systems. The DoD High Order Language (HOL) program was initiated in 1975 with the goal of establishing a single high order computer programming language appropriate for all DoD system development efforts. In 1976, the HOL program became part of an overall program (established by DoD Directive 5000.29) to improve the management of computer resources in major defense systems. A High Order Language Working Group (HOLWG) was established to define the HOL requirements, evaluate existing languages against those requirements, and to implement the minimal set of languages required for DoD use. As a result of HOLWG efforts, DoD Instruction 5000.31 defined a list of seven interim acceptable languages and concluded that none of the languages fully satisfied the initially defined HOL requirements.

The initial requirements were specified in a DoD document entitled STRAWMAN (1975) and evolved through WOODENMAN (1975), TINMAN (1976), IRONMAN (Jan 1977) and revised IRONMAN (July 1977), to the present STEELMAN (1978) document.

The following general HOL design criteria is abstracted from STEELMAN /USDO78/:

- Generality. The language shall provide generality only to the extent necessary to satisfy the needs of embedded computer applications. Such applications involve real time control, self diagnostics, input-output to nonstandard peripheral devices, parallel processing, numeric computation, and file processing.
- Reliability. The language should aid the design and development of reliable programs. The language shall be designed to avoid error prone features and to maximize automatic detection of programming errors.
- Maintainability. The language should promote ease of program maintenance. It should emphasize program readability (i.e., clarity, understandability, and modifiability of programs). The language should encourage user documentation of programs.
- Efficiency. The language design should aid the production of efficient object programs.
- Simplicity. The language should not contain unnecessary complexity. It should have a consistent semantic structure that minimizes the number of underlying concepts. It should be as small as possible consistent with the needs of the intended applications.
- Implementability. The language should be composed from features that are understood and can be implemented. The semantics of each feature should be sufficiently well specified and understandable that it will be possible to predict its interaction with other features. To the extent that it does not interfere with other requirements, the language shall

facilitate the production of translators that are easy to implement and are efficient during translation. There shall be no language restrictions that are not enforceable by translators.

- Machine Independence. The design of the language should strive for machine independence. It shall not dictate the characteristics of object machines or operating systems except to the extent that such characteristics are implied by the semantics of control structures and built-in operations. It shall attempt to avoid features whose semantics depend on characteristics of the object machine or of the object machine operating system. Nevertheless, there shall be a facility for defining those portions of programs that are dependent on the object machine configuration and for conditionally compiling programs depending on the actual configuration.
- Complete Definition. The language shall be completely and unambiguously defined.

Given that none of the interim approved languages satisfied all of the STEELMAN requirements, DoD opted to set forth a program to develop a single language which could, and subsequently funded four contractors to produce competing prototype designs known as GREEN, RED, YELLOW, and BLUE. All four design contractors chose PASCAL as their design point of departure. Design was completed in early 1978 and the GREEN and RED languages were chosen for a one year follow-on design development. On May 2, 1979, the GREEN language (designed by CII-Honeywell Bull) was chosen and renamed Ada.

Since that time Ada has undergone several minor updates resulting in the release in July 1980 of the (proposed standard) Reference Manual for Ada /USDO80b/. Minor revisions will most likely still be required but a fairly stable

definition baseline exists at present. Given this existing baseline the U.S. Army and the U.S. Air Force have recently awarded contracts worth $2.5 million and $9 million (respectively) for Ada compiler development with delivery expected in the 1983-84 time frame.

It is generally agreed that the current definition of Ada, as detailed within the DoD Ada Reference Manual /USDO80b/ has faithfully met the intent of the STEELMAN requirements. It is readily seen that Ada has inherited most of its traits from other modern high level languages. But, unlike most of its predecessors, Ada is designed to be universal, totally portable, and exceptionally reliable. The first stipulation tends to make Ada large and complex - it must support mathematical, process control, or list sorting requirements, as well as system programming requirements. Under the second stipulation, Ada defines its run-time environment so that it executes the same on a microcomputer as it would on a mainframe. But the one trait that was given top priority in Ada design was reliability. An embedded military system obviously cannot tolerate faults during stress periods. Ada must be able to support the development cf easily maintained, very reliable computer programs, and the design of particular Ada features reflects this requirement. Those features which most characterize the Ada rationale are as follows:

- Provide strong data typing facilities with strong type checking to improve software reliability and simplify debugging.
- Provide modularity of program structure to implement nested program units which facilitates information hiding and visibility control.
- Support both top-down and bottom-up design methodologies and provide separate program element compilation capabilities.

- Provide realtime features for parallel processing and scheduling as part of the language definition.
- Provide language defined and user defined exception handling capabilities.
- Provide flexible yet powerful representation features.

If a description of Ada can be summarized within one sentence, it would be that Ada is a versatile general purpose language designed to meet the needs of numerical, scientific, system programming, and real-time applications with an overriding design goal of reducing the cost and improving tne reliability of large scale software development.

2.2 CHILL

The CCITT High Level Language (CHILL) design effort was initiated at approximately the same time as that of Ada. The effort actually indirectly began in 1968 when CCITT Study Group XI undertook the evaluation of over 70 existing high level languages in order to determine if any of them would be suitable for Stored Program Control (SPC) programming of telecommunication circuit switching applications which, at that time, were exclusively programmed in assembly language with all of the attendant disadvantages. Approximately 30 of the original 70 languages were then chosen to be used to conduct extensive programming exercises as part of the evaluation effort. A report called the Yellow Document was released in April 1975 detailing the exercises and their results. The basic conclusion arrived at during this existing language evaluation was that no single language was deemed suitable for SPC programming applications. Also published by CCITT in 1975 was the outline proposal for the CHILL language development which presented the CHILL functional requirements and was known as the GREEN Document.

During the period of 1976 through 1980, CCITT Working Party XI/3 conducted the language definition effort resulting in the release in February 1980 of the proposal for a recommendation for CHILL known as the BROWN Document or Draft Recommendation Z.200. This version was incomplete, however, and a (final) updated version was released in May 1980. This document is numbered COM XI-No. 396 or alternately AP VII-No. 21-E.

CHILL has been the subject of several trial implementations, the first of which was initiated in 1977. Several trial compilers exist or are in progress, although they currently address only language subsets based on preliminary or incomplete definitions.

The features which characterize the CHILL language are almost identical to those of Ada when viewed at a high level. Despite the fact that the original CHILL design objective was limited to development of a language expressly designed for programming SPC telephone exchanges, it can now be seen that CHILL is equally suitable for other general telecommunication applications, as well. It was initially believed by the CHILL designers that SPC programming applications required a High Level Language (HLL) to exhibit some special features not normally associated with an HLL. It was soon discovered that SPC telephone exchange programming was not totally unlike other commonly known real time programming applications, e.g., systems programming, and that certain modern high level languages have been used successfully in these specialized application areas. When viewed in this manner it is seen that the CHILL language design objectives are not very different from those of Ada. In fact, the features listed within the previous section can be duplicated here for CHILL. It is in the interpretation and implementation of these features where the two languages differ and these differences will be discussed in detail within the next section.

SECTION 3

ADA/CHILL FEATURE COMPARISON

3.0 FEATURE COMPARISON

This section will examine the Ada and CHILL programming languages in a dual fashion. The features of the language will be evaluated in terms of their syntatic form, ease of use, availability, etc., and in terms of their function, e.g., if and how well a particular function is implemented and the efficiency of its implementation. The discussion is not intended to be all inclusive but rather to highlight those areas where significant differences exist between the languages in the definition, implementation, or availability of particular features. The following subsections represent major feature categories grouped in this manner for convenience of comparison. The individual language reference manuals, /USDO80b/ and /CCIT80a/, were used as primary sources for material in this section.

3.1 LEXICAL ELEMENTS

The lexical elements of a language represent the smallest identifiable units defined by the language, i.e., the character set, delimiters, identifiers (including reserved words), numbers, character strings, character literals, and comments.

The lexical elements of Ada and CHILL are very similar both in appearance and usage. However, some significant differences do exist as described below.

The character sets used by Ada and CHILL are very similar. Ada uses the standard 128-character ASCII set while CHILL employs the CCITT alphabet No. 5, recommendation V3. The basic character set used to represent CHILL programs is a subset of the basic Ada character set. The differences between the overall sets exist in the representation of the (printable) characters "dollar sign" and "tilde" and in the (non-printable) control character terminology. The binary internal

representation and lexicographical ordering of the two character sets are identical. Note, however, that Ada permits the overloading of any character set through use of the representation specification capability. Additionally, the transliteration facility of Ada allows characters to be represented which are not contained within the basic character set.

Identifiers are the names defined and used within programs. The definition and use of identifiers is almost identical in either language. Identifiers can be built with combinations of letters, digits, and underscores, limited solely by the length of the logical input record. CHILL syntax definition allows multiple repeated underscores and digits in identifiers. This is deemed a significant oversight which could lead to readability problems and fosters unconventional naming/labeling. CHILL distinguishes between identifiers differing only in upper and lower case characters, i.e., CHILL and Chill are two distinct identifiers. Ada does not make this distinction. Both languages have reserved words which may not be used as identifiers. CHILL, however, has a language defined compiler directive ("FREE") which explicitly frees a reserved word for subsequent use as an identifier. Even though explicit use of the compiler directive is a visible clue that reserved words have been freed, this feature is felt to be unnecessary and potentially harmful from a maintenance point of view.

The definition of numeric literals in the languages represents an area of significant difference. Ada defines two classes of numeric literals -- integer and real -- and both integer and real literals can have exponents as well as be represented in any number base between base 2 and base 16. CHILL simply defines integer literals, without exponents, representable in base 2, base 8, base 10 (default), and base 16. While real numbers are not absolutely required in a circuit switching application, there are certain indirectly related applications that make it desireable to have the capability to specify real numbers. For example, it is more convenient (and, in fact, appropriate) to conduct statistical

analysis or accounting tasks if one has access to the set of real numbers. Although off-line reduction and analysis of collected data can be performed by programs written in the more traditional algorithmic languages such as FORTRAN or PASCAL, the significance of collected data is potentially lost at the point of extraction and can never be recovered. If, as is currently proposed, CHILL is to be used in message switching applications as well, this real number exclusion becomes more critical. On the other hand, inclusion of a real number capability creates more difficulties in the area of transportability and representation. However, work is being carried out by IEEE to establish floating point math standards which would reduce these problems.

The representations of character literals and character strings are very similar with one minor difference. Ada allows the non-printable control characters to be used as literals or placed into strings by utilization of the predefined package ASCII. For example, the control characters "carriage return" and "linefeed" would be represented by ASCII.CR and ASCII.LF, respectively. CHILL provides a somewhat similar mechanism using a construct whereby the desired character is specified by a pair of hexidecimal digits which correspond to the lexicographic order of the character in the set. For example, "carriage return" and "linefeed" in CHILL would be C'D0' and C'A0', respectively. While this is equally as effective, it is somewhat more cumbersome and indirect.

Finally, though perhaps not entirely appropriate under the category of lexical elements, there is the definition of compiler directives. Ada supports a wide range of language-defined compiler directives known as pragmas, as well as supporting the creation of implementation-defined directives. CHILL defines only one language-defined directive ("FREE"-reserved word) but also supports implementation-defined directives as well.

3.2 DATA TYPING

This feature category represents the area in which Ada and CHILL most strongly exhibit their Pascal/ALGOL inheritance. A data type defines the set of values which can be assumed by a variable and the operations that may be performed on the variable. The type concept is an abstraction which permits one to ignore the actual values of variables and state that an operation has the effect defined for all values of each given type. Both languages are classified as being "strongly" typed in the sense that they both support strict data typing, data structuring, and compile time (data type) error checking facilities. A good treatment of the justification for associating a type with constants, variables, or parameters of subprograms can be found within the Ada Rationale /ICHB79b/, and is summarized as follows:

- Factorization of Properties, Maintainability. Knowledge about common properties of objects should be centralized and named. Program updates are more convenient since they can be performed at this single central location.
- Abstraction, Hiding of Implementation Details. Implementation details should be hidden from the user. The user need only have knowledge of the external properties of data or program objects.
- Reliability. Objects with distinct properties should be treated in a distinct manner to avoid ambiguity and this distinction can be enforced by the translator.

In order to provide a foundation for presentation of the material contained herein, a high level data type comparison is depicted in Table 3-1. In some cases there is not a direct one-for-one correspondence as might be implied by the chart. Note that the term MODE in CHILL (ALGOL68 derivation) is synonymous with the term TYPE in Ada (PASCAL derivation).

Table 3-1. Ada/CHILL Data Types

Ada Types	CHILL Modes
SCALAR	SCALAR
DISCRETE	DISCRETE
INTEGER	INTEGER
CHARACTER	CHARACTER
BOOLEAN	BOOLEAN
ENUMERATION	SET
(Not Available)	POWERSET*
CONTINUOUS	CONTINUOUS
FLOATING POINT	(Not Available)
FIXED POINT	(Not Available)
COMPOSITE	COMPOSITE
ARRAY	ARRAY
STRING	STRING
RECORD	STRUCTURE
POINTER	POINTER
ACCESS	REFERENCE
DEFINITION	DEFINITION
DERIVED	NEW
SUBTYPE (w/o constraint)	SYNONYM
SUBTYPE (with constraint)	RANGE
(Not Available)	PROCEDURE*

* Not necessarily appropriate to this class, but placed here for convenience.

3.2.1 Type Definition

The method of declaring types within Ada and CHILL is similar in function, though not in form. In both languages, new types are defined in terms of already defined ones by means of type definitions. Both languages offer a set of predefined types (INTEGER, BOOLEAN, etc.) as well as a set of language recognized primitive types (ARRAY, SET, etc.). Using the Ada derived type definition feature (NEWMODE in CHILL), one can create new, logically distinct types having the same properties as the base type. In Ada, if a type is declared in a package specification, the subprograms (including overloaded operators) applicable to the type and declared in the package specification are derived by any derived type definition given after the end of the package specification. CHILL does not support this inheritance of applicable subprograms.

A new mode may be defined by using the SYNMODE feature (subtypes without constraints in Ada) which allows creation of a new mode denotation for the defining or base mode (type renaming). Both Ada and CHILL support combined operations of typing, object declaration, and initialization.

CHILL provides a powerset mode which defines values which are sets of values of an associated member mode. These values range over all subsets of the member mode and CHILL supports the usual set-theoretic operations to manipulate powerset values. Ada does not support a comparable feature within the language definition but does permit a contiguous set of a discrete type to be represented as a range.

Ada provides a real number typing capability wherein the real numbers are approximations of the actual values and which can be represented by the (predefined) floating point type (relative error bound on the value) or the primitive fixed point type (absolute error bound on the value). CHILL does not provide a real number typing capability.

Composite types, i.e., those found by aggregating others, are treated very similarly in the languages. Ada and CHILL both support array, string, and record composite types.

Strings and arrays are handled almost identically but records (structures) in CHILL have two distinct representations: nested structure, which is comparable to the more conventional Ada record representation and level structure, which is derived syntax for a unique nested structure. The level structure mode allows explicit nesting of components within structures as shown by the following example:

```
synmode A = 1,
              2 B bool,
              2 C bool,
                3 D int,
                4 E int;
```

Both Ada and CHILL support the idea of variant structures whereby the values of discriminants are used to define alternative lists of components within a record.

Pointer types are available in both Ada and CHILL. The access type in Ada corresponds to the reference mode in CHILL with both being used in a like manner. CHILL, however, distinguishes between bound reference (access to a location of a given static mode, comparable to the Ada access type) and free reference (access to a location of any static mode). Additionally, CHILL provides a row reference capability which allows definition of reference values for locations of some parameterized mode with statically unknown parameters. In particular, a row value may refer to string locations with statically unknown length, array locations with statically unknown upper bound, or parameterized structure locations with statically unknown parameters.

3.2.2 Type Equivalence

This area has been the subject of many discussions dealing with the relative merits of the typing conventions of modern programming languages. When comparing the features of Ada and CHILL, one of the most relevant issues is the question of name equivalence versus structural equivalence. Name equivalence is based on the principle that every type definition introduces a distinct type. Structural

equivalence, on the other hand, is determined recursively by means of a precise set of rules. It refers to a mechanism whereby some form of equivalence rule is defined between types on the basis of their properties.

Ada employs the name equivalence concept of type equivalence. Two type definitions introduce two distinct types even if they are textually identical. Using the example from the Ada Language Reference Manual /USDO80b/, the objects A and B declared by

```
A: array (1..10) of BOOLEAN;
B: array (1..10) of BOOLEAN;
```

represent two distinct types while the objects C and D declared by

```
C,D: array (1..10) of BOOLEAN;
```

belong to the same type since they are declared via the same type definition. Note that C and D are also distinct from A and B. Ada does, however, allow the user to indirectly create and manipulate types through the use of the subtype definition. Definition of a subtype does not produce a distinct type but rather creates a type which is the same as the parent type except for some (optional) constraint on the value set. This constraint may assume the form of a range, accuracy, index, or discriminant constraint.

CHILL follows the structural equivalence concept though neither name nor structural equivalence are preimposed. The following example illustrates this subtle difference.

```
newmode WEEKDAY = set (MON,TUE,WED,THUR,FRI);
synmode WORKDAY = WEEKDAY;
newmode NOT_WEEKEND = WEEKDAY;
dcl WORK, SICK, VAC NOT_WEEKEND;
    CARPOOL WEEKDAY;
    EARN_MONEY WORKDAY;
```

In this example, WORK, SICK, and VAC are all of the same type, but different from CARPOOL and EARN MONEY. However, CARPOOL and EARN MONEY belong to the same type since the synmode definition only served to rename WEEKDAY to be WORKDAY, not creating a new type. This is similar to the

renaming facility in Ada, i.e., defining a subtype without constraints.

In all the discussions that have been generated concerning the notion of type equivalence it appears to be common that the very arguments one presents against one mechanism are the same arguments for it. For example, in a recent Ada/CHILL comparison by R. T. Boute /BOUT79/, a list of arguments against name equivalence is presented basically stating that it harms program clarity, restricts type manipulation, and undermines program modularity and maintainability. The Ada Rationale uses these exact points to justify rejecting structural equivalence in favor of name equivalence stating ... "We have rejected structural equivalence in order to avoid matching problems for the translator and for the human reader. We also believe that structural equivalence tends to defeat the purpose of strong typing since objects may be considered as being of the same type because their structures are identical by accident, or because they have become identical as a result of textual modification performed during program maintenance. Such objects can then be mixed erroneously without causing translator diagnostics." The argument in the Ada Rationale is a stronger one. The main purpose of employing name equivalence was to restrict type manipulation (mainly for reliability reasons) and lifting that restriction defeats the rationale and most certainly invites programmer abuse.

3.2.3 Parameterization

Another area which is important to the evaluation of a language is the facility for parameterization. Specifically, issues which are usually examined include (1) whether the language provides some form of parameterization for data types and their associated properties and (2) if the evaluation of type parameters is performed entirely at translation time or deferred until execution time.

In Ada, array type definitions can leave index bounds unspecified (unconstrained). These can be subsequently specified by an index constraint for a given array object, so that different array objects of the same type may have different numbers of components. Also, a record type may have variants, i.e., alternative definitions of its components. Different variants are associated with the values of a discriminant component. If the discriminant is constrained, the composition of the record is statically fixed. If the discriminant is unconstrained, the composition of record can be changed during run time by a complete record assignment.

CHILL provides a slightly different data type parameterization facility. There are fixed array and structure modes in which the composition does not change during run time. Also, there are parameterized array and structure modes in which the composition is fixed at the point of creation of the parameterized mode and may not change during run time. Finally, there is the variant structure mode whose composition may change during run time according to the values of certain associated tag fields.

It can be seen that the type parameterization facilities in Ada and CHILL overlap in most respects. The one advantage held by CHILL is the language-defined ability to dynamically change the composition of a structure. This is potentially cumbersome in Ada (employing complete record assignment) if the record structure is complex.

Another aspect of parameterization is whether a language allows types and procedures to be typed and hence, treated in the same fashion as other objects, e.g., passed as parameters to functions or procedures.

Types and procedures are not typed in Ada and thus cannot be passed as parameters to functions or procedures. However, the Ada generic clause provides a general facility for translation time parameterization of program units. A generic clause permits parameterization of the text of a package or of other program units. Replication of text can thereby be avoided, promoting readability. Also, the

translator may use its knowledge of data type representations to achieve certain optimizations. Seen in this light, the generic facility provides a natural complement to strong typing /ICHB79b/.

CHILL allows procedures to be defined as modes and allows them to be passed as parameters to other procedures. Procedure modes in CHILL thus allow procedures to be handled in exactly the same manner as other variables.

3.2.4 Representation Control

One of the most important features a high level language must possess in a systems environment is the ability to provide an efficient means of mapping the software onto the hardware. This potentially contradicts the notion of generality in terms of having to deal with specific physical representations. And, it goes against the stated objectives of HOL implementation whereby data typing and abstraction are encouraged. However, by providing language features which allow explicit control over the physical mapping, efficient (though less machine independent) software can be generated, and this is an equally critical objective.

Both Ada and CHILL provide adequate representation control capabilities, and both associate the representation specifications with the type rather than with individual objects of the type.

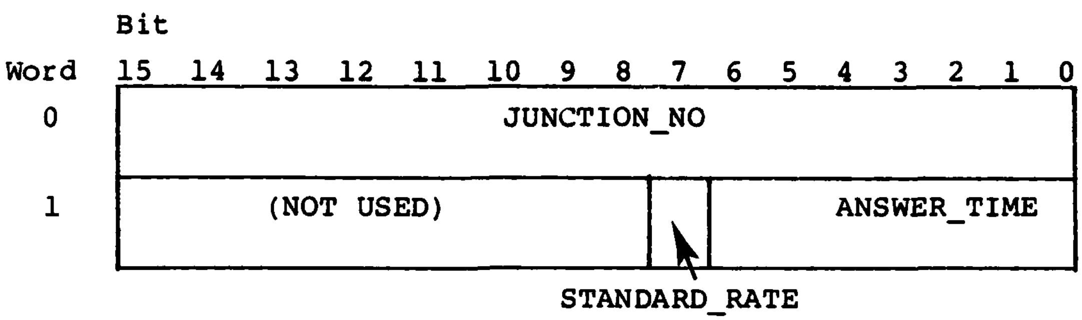
CHILL provides explicit layout control of both structure and arrays. For structures, the positions of fields may be described in terms of word and bit positions. For arrays, the step specification indicates the position of the first element and the number of bits allotted to each element in the array.

Ada allows enumeration (set) type representation specification, which CHILL does not. Ada provides for array type layout control when embedding the array within a record, and provides explicit record layout control capabilities which are very similar to CHILL's. However, the syntactic structure of the Ada representation construct is considered cleaner and

easier to use than that of CHILL, as seen in the following example of a record layout which similarily maps onto the same machine. In CHILL, assuming 16-bit words, the following declaration:

```
dcl CALL_RECORD struct (JUNCTION_NO int pos(0),
                        ANSWER_TIME int (0:100) pos (1,0:6),
                        STANDARD_RATE bool pos (1,7));
```

produces the following binary layout:



In Ada, assuming the type definition had already been elaborated, the equivalent form is:

```
    for CALL_RECORD use
                        record;
                          JUNCTION_NO at 0 range 0..15;
                          ANSWER_TIME at 1 range 0..6;
                          STANDARD_RATE at 1 range 7..7;
                        end record;
```

which produces the same internal mapping as the CHILL example. It can be seen, however, that the Ada example is more readable and explicit in its presentation.

3.3 NAMES, EXPRESSIONS AND STATEMENTS

Names are used to reference declared entities, expressions are formulas that define the computations of particular values, while statements constitute the algorithmic part of a particular program. This section presents information on how Ada and CHILL define and manipulate named entities and how expressions and statements are used.

3.3.1 Names and Expressions

Names and expressions are handled almost identically in both languages. Array component indexing and record field selection are performed in like fashion in either language, using paranthetical and dot notation, respectively. Slices of one dimensional arrays (or strings) may be specified with CHILL allowing a slice (subarray/substring) to be declared using either a range (as in Ada) or a start position and length.

Aggregates (tuples) may be formed from array or record component values. Ada and CHILL both allow aggregate construction using either positional and/or named assignment. In Ada, these two forms of assignment may be mixed in any one aggregate specification, while CHILL requires utilization of one method or the other in any one specification. CHILL provides for powerset aggregate specification as well as array and structure aggregates.

Expressions are formed in analogous fashion in either language with slight differences reflected in the availability and usage of operators. Ada provides short circuit control operations (with the same precedence as logical operators) which provide additional control over expression evaluation by "short-circuiting" a potential exception causing condition. This feature fosters program integrity at execution time as shown in the following example:

```
if I/=0 and then A/I=B then
      .
      .
      .
end if;
```

Without the short circuit operator "and then", a run time exception would occur on the attempt to divide by zero in the second term of the expression above.

Both languages provides membership operators and, in addition, CHILL supports a wide range of language-defined, set-theoretic operators in conjunction with its powerset mode feature.

Finally, Ada allows most operators to be redefined. This "overloading" of an operator is used to hide the declaration of another operator as well as to provide local explicit control over operator utilization. Ada's ability to redefine operators through overloading coupled with the capability to define data types in package specifications allows Ada to be extended in a safe manner through data encapsulation. For example, operator overloading would permit one to define arithmetic operations on very short or very long objects in an efficient manner. CHILL does not support the concept of operator overloading, and this is viewed as a deficiency.

3.3.2 Statements

The action statements that are available within each language are comparable in both form and function. The assignment, exit, return, goto, and if statements are all handled in very similar fashion.

The CHILL loop control statement provides three different forms: the traditional "do loop", the "do while", and the "do with", which is used as a shorthand notation for accessing structure fields. The loop construct in CHILL allows non-unitary increments in both a forward and reverse direction. Ada supports only forward and backward unitary increments. Both languages define an infinite loop feature.

The basic case statement features of Ada and CHILL are comparable. However, CHILL extends the case concept to include a decision table case statement in which complicated conditions can be expressed in tabular form. Ada does not support a comparable feature. An example from /NCSY80/ illustrates this feature.

```
module
  dcl I read int:=ININT(),
      C read char:=INCHAR(),
      B read bool:=INBOOL(),
      X int;
```

```
case I,        C,           B,           of
     (1),      ('A'),       (TRUE):      X:=1;
     (2:5),    ('D':'F'),   (FALSE):     X:=2;
     (else),   ('G':'Z'),   (*):         X:=3;
     else                                X:=4;
esac;
end;
```

This example basically says, if cases I, C, and B are all true in each subsequent line, appropriate assignment to X takes place. Note that the asterisk is used as a "don't care" value. Note also the multiple (and hence, ambiguous) usage of the colon.

While Ada does not support a comparable feature in the language definition, the equivalent form of the above example in Ada (assuming the same variables) is as follows:

```
   if I =      1    and C = 'A'      and  B=TRUE  then  X:=1;
elsif I in     2..5 and C in 'D'..'F' and  B=FALSE then  X:=2;
elsif I not in 1..5 and C in 'G'..'Z'              then  X:=3;
else                                                     X:=4;
end if;
```

All statements can be labeled in Ada while CHILL restricts statement labeling to bracketed action statements or statements with named handlers for errors. Note that Ada distinguishes between a label and a loop identifier in that the latter is not a label but an aid in viewing program structure. It should also be noted that CHILL allows a statement to have a handler appended in order to take care of possible exceptions caused by statement execution. Ada permits case statements or even arms of case statements to have their own exception handler and, in fact, permits appending an exception handler to any statement through the use of a begin block. And, as a final note, CHILL uses the backward opening bracket name as the closing bracket on compound statements (e.g., CASE...ESAC) while Ada employs the more readable closing bracket mechanism (i.e., CASE...END CASE).

3.4 PROGRAM STRUCTURE

Within this feature category are several issues key to the goal of defining a reliable, maintainable programming language. These issues are addressed within the concepts of modularity, scope, and visibility. The following subsections will address these areas and how they are handled within Ada and CHILL. These areas tend to overlap leading to some recursiveness in discussion.

3.4.1 Modularity

One of the more popular topics associated with software engineering and programming languages in recent years is the concept of modularity. Building programs through the use of modules allows the progammer to group logically related items. The ability to package declared entitites, such as subprograms, data elements, types, and other modules provides a powerful structuring tool for complex programs.

Ada supports modules called packages. Packages may have two textually distinct parts which can be separately written and compiled; a package specification, which determines the resources made available by a package to the user, and a package body, which implements the resources provided by the package. The declaration (specification) and the implementation (body) are well separated. In fact, the specification represents the complete interface definition for the programmers using the package, for the implementation of the package body, and for separate compilation.

Ada considers three uses of packages /ICHB79b/:

(1) Named collections of declarations: logically related variables, constants, and types to be used in other program units.

(2) Groups of related subprograms: logically related functions and procedures which share internal "own" data, types, and subprograms.

(3) Encapsulated data types: definition of new types and associated operations in such a way that the user does not have knowledge of the type's internal properties.

Named collections of declarations can most closely be associated with the idea of system level common data. In fact, this type of package can be likened to named common in FORTRAN, with the exception that types as well as objects may be declared. The following example shows how groups of logically related entities can be meaningfully grouped:

```
package CALL_SIGNALS is
   ON_HOOK,METERING:boolean;
   SUBS_ID,DIGIT:integer;
   type SIGNAL is (D_TONE,R_TONE,R_SIG);
end CALL_SIGNALS;
```

Accessibility to objects declared within the above package is obtained by dot notation (as with record component selection) or by a use clause, as shown below:

```
declare
   use CALL_SIGNALS;
begin
   ON_HOOK:=FALSE;
end;
```

The grouping of related subprograms can be likened to the concept of a subroutine library. Typically, the package will contain a visible part where declarations of the contained subprograms reside, and a hidden part where the actual subprogram bodies and local data reside. The separation of the two parts is clear and distinct. In general, the two parts need not be textually contiguous and can be compiled separately - providing protection for and physically hiding the package body:

```
package LOCAL_CALL is
   procedure RCV_SIGNAL(ON_HOOK:out boolean);
   procedure SND_SIGNAL(A:in SIGNAL);
   procedure MAINTENANCE(IN_SERVICE:out boolean);
end;
```

```
package body LOCAL_CALL is
   type STATE is (IDLE,OFF_HOOK,RINGING,OUT_OF_SERVICE);
   procedure METERING (SUBS_ID:in INTEGER) is
   begin
      -- perform call metering tasks
   end;
   procedure RCV_SIGNAL(ON_HOOK:out boolean) is
   begin
      -- perform signal recognition tasks
   end;
   procedure SND_SIGNAL(A:in SIGNAL) is
   begin
      -- perform signal sending tasks
   end;
   procedure MAINTENANCE(IN_SERVICE:out boolean) is
   begin
      -- perform maintenance processing tasks
   end;
end LOCAL_CALL;
```

In the above example, the three procedures declared in the package specification are visible while procedure METERING is hidden. Note, however, that METERING is visible to the three procedures within the package.

Excapsulated data types correspond to a situation in which we want the name of a type to be public, but where the knowledge of its internal properties is to be available only to the subprogram bodies contained in the module body /ICHB79b/. The type name is specified within the visible part of the package along with the specification that the type is "private". The full definition of the type then follows within a hidden private part:

```
package CALL_SIGNALS is
   type SUBS_ID is private;
   ON_HOOK,METERING:boolean;
private
   type SUBS_ID is new INTEGER range 0..9999;
end;
```

The three forms of modules or packages described above can be used in the traditional manner to construct libraries containing common pools of data and types, application packages, and complete systems.

Additionally, Ada provides the capability to parameterize modules by means of generic clauses. Generic program units can be viewed as models or templates for other variant program units and expansion of the generic unit at translation time has the effect of creating a named instance (copy) of the unit. According to the Ada Rationale /ICHB79b/ the objectives in providing the generic program unit capability were as follows:

1) Allow additional freedom of factorization without sacrificing efficiency
2) Minimize the amount of code presented to the translator
3) Preserve regular program unit security
4) Introduce a modest language extension with minor impact

This feature is considered to be very useful in avoiding wasteful replication of text while yielding better readibility. Also, it is possible for a translator to use its knowledge of instantiated data representation to optimize space allocation when data is to occupy the same amount of space in the same representation.

CHILL supports two kinds of modular structures called modules and regions. Regions are similar to modules in form but are associated with processes and concurrency and will be addressed later. The module as defined in CHILL is similar in function to the package in Ada. However, in CHILL it is not possible to separate the specification part from the implementation part. This is seen as a definite liability which restricts the modularity of the CHILL language and limits the ability to effectively follow a top-down design approach.

The following example shows a CHILL module that is analogous to the previous Ada package:

```
LOCAL_CALL:
module
   newmode STATE=set(IDLE,OFF_HOOK,RINGING,OUT_OF_SERVICE);
   METERING:
   proc (SUBS_ID int in);
      /*perform call metering tasks*/
   end METERING;
   RCV_SIGNAL:
   proc (ON_HOOK bool out));
      /*perform signal recognition tasks*/
   end RCV_SIGNAL:
   SND_SIGNAL:
   proc (A SIGNAL in);
      /*perform signal sending tasks*/
   end SND_SIGNAL;
   MAINTENANCE:
   proc (IN_SERVICE bool out);
      /*perform maintenance processing tasks*/
   end MAINTENANCE;
end LOCAL_CALL;
```

The above module must be compiled as a unit - there is no separation of item declaration from its associated body. It is felt that the Ada package concept is superior in terms of modularity and separate compilation. CHILL also does not support any feature comparable to Ada's generic feature, and this is considered a drawback.

The Ada package represents one of three forms of program units of which Ada programs can be composed. The other forms are tasks (discussed later) and subprograms.

In Ada there are two forms of subprograms: procedures and functions. A procedure call is a statement; a function call returns a value. The specification of a procedure specifies its identifier and its formal parameters (if any). The specification of a function specifies its designator, its formal parameters (if any), and the subtytpe of

the returned value. All Ada subprograms can be called recursively and are reentrant.

The formal parameters of an Ada subprogram are considered local to the subprogram and can assume one of three modes:

IN	The parameter acts as a local constant which obtains its value from the actual parameter.
OUT	The parameter acts as a local variable whose value is assigned to the actual parameter upon subprogram execution.
IN OUT	The parameter acts as a local variable, permitting access and assignment to the actual parameter.

Scalar or access type parameters are passed by value (actual parameter copied into formal parameter and vice versa, as appropriate) upon subprogram call. Array, record, or private types may be copied, or alternately, the formal parameter may only provide access to the actual parameter during subprogram execution (pass by reference or location). Ada does not define which mechanism is to be employed for parameter passing. This could potentially result in inefficient parameter passing if the particular implementation does not optimally choose the appropriate mechanism for the parameters being passed.

CHILL does not distinguish between a procedure and a function in a true sense. Instead, the procedure definition dictates whether it is to be used as a value returning procedure (function) or as a normal procedure.

As in Ada, the formal parameters of a CHILL procedure are considered local to the procedure and can assume one of three modes - IN, OUT, or IN OUT.

The storage and manipulation of the formal parameters in relation to their local usage is nearly identical to Ada. One exception is that the mechanism for passing parameters by value or by location can be explicitly specified in CHILL. This can result in inefficient implementation if one neglects to specify the pass by location mechanism for large

objects. It could also lead to maintenance-related problems if the size of a particular parameter is changed and the programmer neglects to also change the passing mechanism specification to match the data structure being passed.

3.4.2 Scope and Visibility

This subject was touched upon in the previous section and will be addressed further herein.

A declaration associates an identifier with a program entity such as a variable, a type, a subprogram, a formal parameter, or a composite structure component. The region of text over which a declaration has an effect is called the scope of the declaration. An entity declared immediately within a unit is said to be local to the unit; an entity visible within but declared outside the unit is said to be global to the unit. A closed scope is one where only the external objects that have been explicitly indicated by a visibility expansion clause are visible. This is the most restricted and hence the most secure interpretation. An open scope implies that identifiers declared in outer contexts are automatically visible in inner nested contexts unless an explicit visibility restriction is given.

Ada follows an open scope policy as the default option in its definition. The rationale for this decision is that (1) the lists of explicit visibility expansion clauses would grow to unmanageable lengths and (2) the programmer would tend to use "standard" (all-inclusive) lists anyway. The visibility rules provided in Ada combine a traditional visibility inheritance mechanism with the ability to explicitly control the set of names that can be accessed within a given program context. This ability follows from the naming conventions and the previously mentioned module facility and visibility restrictions. A renaming capability is also provided to assist in resolving name conflicts. As an additional syntactic convenience, a USE clause mentioning names of visible packages may appear in the declarative part. The

effect of the USE clause is to cause certain identifiers of the visible parts of the named packages to become directly visible.

CHILL, on the other hand, applies the same open scope rules for blocks and procedures, but restricts the scope of a module, i.e., no identifiers are automatically inherited. Names declared in a module are local to that module. However, global names, i.e., names declared outside the module, are not automatically visible inside the module. Furthermore, local names of a module may be made visible outside the module. To make a global name visible within a module, the name must be mentioned in a "seize" statement. To make a local name visible outside a module, the name must be mentioned in a "grant" statement.

3.5 CONCURRENCY

Concurrent processes are those which overlap in time. They are called disjoint processes if they do not interact and interacting or cooperating processes if they do. Much has been written on the subject of concurrency and it represents an area which attracted considerable attention during Ada and CHILL definition activities. Obviously, concurrent processes model the activities which occur within many embedded computer applications. This section will examine the tools provided within Ada and CHILL for handling concurrency. No attempt will be made to address the nature of the implementation necessary to support the features, as this is outside the scope of this report.

The rationale employed for definition of the concurrent processing (tasking) facility in Ada is that the traditional semaphores, events, and signaling mechanisms are clearly at too low a level and individually exhibit too many drawbacks. Monitors /BRIN73/ on the other hand are too difficult to understand, awkward to use, and an unfortunate mix of low level and high level concepts /ICBH79b/.

The Ada design philosophy was to strike a balance between the low level and the high level controlling mechanisms while providing a simple powerful tool. It appears that the designers achieved their goal.

The task represents the basic parallel processing structure within the Ada language. Structurally the task is analagous to the Ada package. Communication and synchronization between executing tasks is provided by using the concept of a rendezvous between a task issuing an entry call and a task accepting the call by an accept statement. Thus, both the "caller" and the "callee" must be present at the rendezvous for synchronization and/or communication to occur. Subsequent tasks calling a currently executing task are suspended, queued, and handled on a first-in, first-out basis. The priorities of tasks in the system are assigned at compile time using the pragma PRIORITY. The effect of priorities on scheduling is defined by the following rule: If two tasks with different priorities are both eligible for execution and could sensibly be executed using the same processing resources, then it cannot be the case that the task with the lower priority is executing while the task with the higher priority is not.

Tasks may be created by (1) defining a task type that indicates a general specification from which objects may be created or (2) a single task declaration which is equivalent to using an anonymous task type. The ability to specify task types offers roughly the same advantages associated with generic packages, as described previously.

A task body defines the execution of the tasks of the corresponding type. The activation of a task object consists of the elaboration of the declaration part, if any, of the corresponding task body. After activation, the statements of the task body are executed. Normal termination of a task occurs when its execution reaches the end of its task body and all dependent tasks, if any, have terminated. Abnormal termination can be forced by means of an abort statement.

Further flexibility is provided by the select statement which allows a calling or called task to select from a set of alternatives at the point of rendezvous. The select statement can assume three forms:

- Selective wait by the called task
- Conditional entry by the calling task
- Timed entry by the calling task

The following buffering task example taken from the Ada reference manual /USDO80b/ illustrates the Ada tasking facility. Assume there is a producer task outputting characters until an EOT is encountered and a consumer task inputting characters until receipt of the EOT:

```
task BUFFER is
   entry READ(C:out CHARACTER);
   entry WRITE(C:in CHARACTER);
end;

task body BUFFER is
   POOL_SIZE:constant INTEGER:=100;
   POOL:array(1..POOL_SIZE) of CHARACTER;
   COUNT:INTEGER range 0..POOL_SIZE:=0;
   IN_INDEX,OUT_INDEX:INTEGER range 1..POOL_SIZE:=1;
begin
   loop
      select
        when COUNT < POOL_SIZE= >
           accept WRITE(C:in CHARACTER) do
             POOL(IN_INDEX):=C;
           end;
           IN_INDEX:=IN_INDEX mod POOL_SIZE + 1;
           COUNT:=COUNT + 1;
      or when COUNT > 0= >
           accept READ(C:out CHARACTER) do
             C:=POOL(OUT_INDEX);
           end;
           OUT_INDEX:=OUT_INDEX mod POOL_SIZE + 1;
           COUNT:=COUNT - 1;
```

```
         or
            terminate
         end select;
      end loop;
   end BUFFER:
```

The Ada definition of tasks is consistent with the state of the art philosophy of handling the concurrency concept. In fact, Ada's approach closely resembles recent proposals by Brinch Hansen /BRIN78/ and Hoare /HOAR78/.

CHILL offers a range of features to handle synchronization and communication between cooperating processes. The CHILL analogy to the Ada task is the process, though it is more similar to the process of concurrent Pascal. CHILL also provides regions and events which are similar to the concurrent Pascal monitors and queues, respectively.

A CHILL process is textually similar to, but semantically different from, a CHILL procedure. Process instances can be created and activated by means of a start statement. When a process is activated by a start statement, actual parameters may be passed to the activated task at activation time. The CHILL instance mode is similar to the Ada task type. Like Ada, a process may terminate itself (via a stop statement) or terminate normally. The operations defined for instance modes are equality and the parameterless procedure "THIS" which yields the instance value of the process invoking it. Ada does not have these features.

The CHILL region is the means of providing mutual exclusion. Regions correspond to modules and all previous remarks dealing with CHILL modules apply here. Critical procedures are procedures which are defined within regions.

There are also several synchronizing primitives defined in CHILL. Events are provided which facilitate process synchronization. It is possible to delay a process to make it wait for an event to occur, and a process may cause an event to occur such that delayed processes are able to continue. A delayed process becomes a member (with a priority) of a set of delayed processes attached to a specified event location. The

delay statement allows the optional process priority to be specified. Upon execution of the corresponding continue statement, the process with the highest priority associated with the particular event is selected to become active according to an implementation-defined scheduling algorithm. A delay case statement is provided which allows a process to wait for one of a number of events. Buffer mode objects and their operations are used to provide communication between processes. Messages can be sent to and received from buffers by processes through the use of send and receive constructs. Also, there are CHILL signals. Signals are used to provide both synchronization and communication. A feature called the receive case statement allows the receipt of any one of a set of buffers or signals and is similar to the delay case statement in form but with added facilities to handle the message part of buffers or signals.

The following call queuing example taken from the CHILL reference manual /CCIT80a/ illustrates the CHILL tasking facility:

```
SWITCHBOARD:
   module
   dcl OPERATOR_IS_READY,
      SWITCH_IS_CLOSED event;

   CALL_DISTRIBUTOR:
   process();
      do for ever;
         wait(10 /*seconds*/);
         continue OPERATOR_IS_READY;
      od;
   end CALL_DISTRIBUTOR;
```

```
    CALL;
    process();
       delay case
       (OPERATOR_IS_READY): /*some action*/ ;
       (SWITCH_IS_CLOSED): do for i int(1:100);
                             continue OPERATOR_IS_READY;
                             /*empty the queue*/
                          od;
       esac;
    end CALL;

    OPERATOR:
    process();
       do for_ever;
          if TIME = 1700
             then
                continue SWITCH_IS_CLOSED;
          fi;
       od;
    end OPERATOR;

    start CALL_DISTRIBUTOR():
    start OPERATOR();
    do for i int(1:100);
       start CALL();
    od;
 end SWITCHBOARD;
```

It is readily seen that CHILL provides a wide selection of tools to handle concurrency. This is seen as a disadvantage by some. One author says, "The CHILL approach indicates the 'if in doubt, put it in' attitude of the language designers. This has resulted in a heterogeneous collection of mechanisms, for which it is difficult to develop a unified program design and analysis model. There is also a high degree of redundancy, for example, the "buffer" can be easily implemented by regions and events (classical concurrent Pascal

example) and vice versa. The obvious indecision has resulted in a poor overall design." /BOUT79/

The Ada approach on the other hand is concise and powerful. There appears to be a minimum amount of redundancy in design.

3.6 EXCEPTION HANDLING

Exceptions can be categorized as either errors or infrequent (non-normal) events and there are many schools of thought as to the mechanisms that should be employed to handle exceptions. It is generally agreed, however, that a facility for handling exceptional conditions is essential for reliability of real time systems. In many cases, systems must be designed to continue to function (though perhaps in a reduced capability configuration) through hardware or software casualty situations. This is especially true for embedded military weapons and communications systems where failure in time of stress could have serious consequences. Ada and CHILL both have extensive exception handling features which are very similar in form and function.

In Ada, there are both user-defined exceptions and predefined exceptions. Exceptions may be recognized automatically (i.e., the predefined exceptions are raised when the indicated error conditions arise) or by the user by executing the "raise" statement. When an exception has been raised, the execution of the program is stopped at that point and processing proceeds at the appropriate exception handler.

The exception handler is the mechanism which provides the executable code in response to a named exception. In Ada, the handler appears at the end of a block or of a body of a subprogram, a package, or a task. As previously stated, Ada permits case statements (and arms of case statements) to have their own exception handler and, in fact, permits appending an exception handler to any statement through the use of a begin block. Note that the handler is a substitute for retaining the code at the point an exception is raised. In Ada, the syntactic form of the handler is similar to the case statement.

Ada provides a compiler directive which may be used to suppress some exceptions. This suppression may apply to all appropriate operations, all appropriate operations on a given type, or all appropriate operations on a given object.

Special attention is given to exception handling in parallel Ada tasks. Restrictions are placed on propagation of an exception from one task to another. In general, exceptions are propagated during rendezvous (i.e., intertask communication). A task may explicitly raise the failure exception in any other visible task.

The CHILL exception handling facilities are nearly identical to those of Ada and as such need not be elaborated. The only items which should be pointed out are as follows:

- CHILL does not support the explicit suppression of exceptional conditions.
- CHILL allows an exception to be directly appended to a statement. This can be helpful but can also lead to readability problems, if abused.
- CHILL allows an exception list to be specified in a procedure definition indicating which exception can be propagated to a caller. This is useful in the case where an exception is not explicitly specified in the current unit and propagation must occur.

3.7 INPUT/OUTPUT

"No standard input and output routines are defined in CHILL. Such routines may be written in CHILL itself." /NCSY80/ Unfortunately there is no information in either the CHILL introduction /NCSY80/ or the CHILL definition document /CCIT80a/ to confirm or deny the second statement above. Therefore, this section can only address the I/O features within the Ada language definition.

The Ada Rationale states the problems associated with language defined I/O features very well. "... the needs for application level input-output may vary greatly between classes of applications. For example, file manipulation, batch processing, line and page layout, interactive input, and non-character processing pose significantly different problems. An attempt to build in special features to cover the range of input-output applications would mean that every user and every translator would be forced to take account of this additional complexity. A major design goal in the ... language was therefore to provide the ability to develop a rich set of input-output facilities without additional language constructs." /ICHB79b/

Three standard input-output packages are provided in the Ada language definition.

The generic package INPUT_OUTPUT defines a general set of user level I/O operations. These operations are applicable to files containing elements of a single type - e.g., character files, integer (binary) files. General operations which are provided for file manipulation include file creation, OPEN/CLOSE file commands, NAME file commands in addition to traditional file I/O operations (e.g., READ, WRITE, EOF).

Additional operations for text related I/O are defined in the second standard package, TEXT_IO, which is defined in terms of the package INPUT_OUTPUT. Basically, TEXT_IO provides facilities to perform file I/O in human readable form.

Finally, the package LOW_LEVEL_IO defines the form of the operations used when dealing with low level I/O to a physical device. Such operations are handled by using one of the predefined procedures SEND_CONTROL and RECEIVE_CONTROL. These procedures are declared in LOW_LEVEL_IO and have two parameters which identify the device and the data. However, the kinds and formats of these control parameters will depend on the physical characteristics of the particular device.

3.8 DISCUSSION

The above material has shown the technical similarities of the two langages. In no feature category does the CHILL language exhibit any distinct linguistic or functional advantage over Ada. Certainly there are minor tradeoffs seen in the form or usage of a particular construct. But the overall feature comparison has uncovered no distinct technical advantage in using CHILL over Ada for SPC circuit switching applications.

Additionally, no information was provided within the CHILL language definition document as to the nature and extent of the facilities for (1) Input/Output, (2) in-line machine code insertion, or (3) interface to "foreign" code. These are three very critical areas within the context of circuit switching software applications, and the fact that the form and function of these (somewhat machine dependent) facilities were not addressed within the language definition is extremely disconcerting.

Ada provides facilities for language defined I/O packages, defines a mechanism for machine code insertion, and allows Ada programs to interface to programs written in other languages (for example, CHILL).

CHILL does not support any language defined Input/Output features. Also, no evidence could be found that the language definition supports machine code insertion or a foreign code interface mechanism. One can perhaps argue that these features are not necessary and hence, better left out. However, the fact is that in certain situations, having access to the capability to write (and execute) in-line machine code can be a valuable tool for space and time optimization. A similar argument can be made for the ability to interface to foreign code, where optimization in the other language might be required for efficient implementation. The exclusion of the ability to support these features within the language definition is considered a rather major oversight.

SECTION 4

PROGRAMMING ENVIRONMENT EVALUATION

4.1 CHILL/SDL/MML ENVIRONMENT

This section will present overviews of the Specification and Description Language (SDL) and the Man-Machine Language (MML), and discuss their overall relationship to CHILL.

"SDL is a means of representing the specification of the functional requirements and also the description of the logic processes necessary to implement the specification, in stored programme control (SPC) switching systems." /CCIT80b/. The method of presentation is based on state transition diagrams.

The main areas of application cover all types of SPC switching systems. Within these systems examples of processes which can be documented using SDL are: call processing (e.g., call handling, routing, signalling, metering, etc.), maintenance and fault treatment (e.g., alarms, automatic fault clearing, configuration control, routine tests, etc.) and system control (e.g., overload control).

The requirements of a system are defined in the specification of that system and the implementation of those requirements is defined in the description of that system.

The objective of the SDL is to provide a standardized method of presentation that /CCIT80b/:

- Is easy to learn, to use, and to interpret in relation to the needs of operational organizations.
- Provides unambiguous specifications and/or descriptions for tendering and ordering.
- Provides the capability for meaningful comparisons between competitive types of SPC switching systems.
- Is open-ended to be extended to cover new developments.

To meet these objectives two forms of the SDL have been developed. The graphical form, SDL/GR, is a method whereby each process is represented in terms of states and the transitions between them. An input causes the process to leave a state and travel along a transition executing tasks, generating output signals, and branching on decisions until another state is reached. The representations may be linear, with multiple appearances of a single state if convenient, or may be of mesh form or any combination of the two. The concepts of state, input, task, output, decision, and save are represented by their respective symbols. The appropriate interconnection of such symbols by flow lines represents the logical flow of a process. Strict rules for drawing sequence, flow, and annotation are applied. There is no correlation between the graphical form of SDL and the CHILL programming language.

The other form of SDL is the program-like form SDL/PR, previously known by the more descriptive name, Machine Readable Form (MRF). The SDL/PR is intended to facilitate the automatic generation, modification, and analysis of SDL diagrams. SDL/PR is still in the development stage at this time.

Much effort has been expended in the determination of the correlation requirements for CHILL and SDL. In fact, CCITT Study Groups XI/3-1A (MRF subgroup) and XI/3-1B (SDL/HLL subgroup) devoted much time in 1978 to this very question and concluded ". . . a strong correlation between SDL and CHILL is now not only possible but also more likely to occur in actual implementation." /CCIT78/ This conclusion enabled them to justify the disbanding of the separate subgroup for ensuring correlation. Since that time however, less emphasis has been placed on the requirement for correlation. In fact, at present there is no correlation between SDL/PR and CHILL beyond the obvious similarities of their form and application domain.

The Man-Machine Language (MML) is used to facilitate operation and maintenance functions of SPC switching systems of different types. According to different national requirements, MML can also be used to facilitate installation and testing of such systems /CCIT80c/.

The MML contains inputs (commands), outputs, control actions, and procedures sufficient to ensure that all relevant functions for the operation, maintenance, installation and testing of SPC systems can be performed. It has been designed with an open ended structure such that any new function or requirement added will have no influence on the existing ones. The language structure is such that subsets can be created which may be necessary for administrative or implementation reasons.

The MML is a totally independent tool which is not correlated with CHILL in any sense other than the fact that they may share the same application environment in a (mutually) cooperative manner.

4.2 ADA PROGRAMMING SUPPORT ENVIRONMENT (APSE)

This section presents an overview of the Ada Programming Support Environment (APSE) taken from the STONEMAN document /USDO80a/.

The overall objective of an APSE is to offer cost-effective support to all functions in a project team engaged in the development, maintenance and management of a software project, particularly in the embedded computer system field, throughout the lifetime of the project.

An APSE adopts a host/target approach to software construction. That is, a program which will execute in an embedded target computer is developed on a host computer which offers extensive support facilities. Except where explicitly stated otherwise, this document refers to an APSE system running on a host machine and supporting development of a program for an embedded target machine.

An APSE offers a coordinated and complete set of tools which is applicable at all stages of the system life cycle, from initial requirements specification to long-term maintenance and adaptation to changing requirements.

The tools communicate mainly via the database, which stores all relevant information concerning a project throughout its life cycle. The database is structured so that relationships between objects in the database can be maintained, in order that configuration control problems can be resolved.

Individual functions supported by the tools in an APSE include:

- Creation. It is possible to create database objects which contain specifications, design documentation, program source text, program documentation, test data, and so on.
- Modification. A database object can be modified to produce a new object (or a new version of the same object), for example, by editing.
- Analysis. The entities in a database object can be analyzed, producing a new object which records the results of this analysis. Examples of such analysis are set/use and cross reference listings.
- Transformation. The representation of a database object may be changed by transformation tools.
- Display. Objects can be displayed on terminals, printers, and so on.
- Linking. A collection of compiled code objects can be consolidated, resulting in a new object ready for loading and execution.
- Execution. Once a program has been compiled and linked, it can be loaded and executed, possibly with an appropriate environment being

used to supply test information and to monitor execution.

- Maintenance. The APSE must enable configuration control to be maintained. For any configuration of software, it is necessary to be able to determine the origin and purpose of each component of the configuration and to control the process of further development and maintenance of the configuration.

The user interface offered by an APSE is independent of the host machine.

At all stages of the development of a program - design, coding, testing, maintenance - an APSE encourages the programmer to work in Ada source terms, rather than in terms of the assembly language of the particular host or target machine.

Extension of an APSE toolset requires knowledge only of the particular APSE and of the Ada language. A new tool - for example, an environment simulator - is written within the APSE. This tool can then be installed as part of the APSE and subsequently invoked.

An APSE supports the use of libraries of standard routines for incorporation in programs written for both host and target machines.

The above paragraphs outline the facilities offered by an APSE to its users in support of Ada programming. However, a further requirement is for portability both of APSE tools between, for example, APSEs hosted on different machines and of complete APSE toolsets. To address this aim and to indicate a means of implementation of an APSE designed to provide portability, this document gives requirements for a low level portability interface and support function set (the KAPSE) together with a minimal toolset (the MAPSE).

The purpose of the KAPSE is to allow portable tools to be produced and to support a basic machine-independent user interface to an APSE. Essentially, the KAPSE is a virtual support environment (or a "virtual machine") for Ada programs, including tools written in Ada.

The declarations which are made visible by the KAPSE are given in one or more Ada package specifications. These specifications will include declarations of the primitive operations that are available to any tool in an APSE. They will also include declarations of abstract data types which will be common to all APSEs, including the data types which feature in the interface specifications for the various stages of compilation and execution of a program.

While the external specifications for the KAPSE will be fixed, the associated bodies may vary from one implementation to another. In general all software above the level of the KAPSE will be written in Ada, but the KAPSE itself will be implemented in Ada or by other techniques, making use of local operating systems, filing systems or database systems as appropriate.

The minimal APSE (MAPSE) is one which provides a minimal but useful Ada programming environment and supports its own extension with new tools written in Ada. Hence, the MAPSE is an APSE and must meet the general requirements set down for APSEs.

For many important activities during a project life cycle as listed below, the only support offered by the MAPSE consists of general text manipulation facilities. A more comprehensive APSE will offer specialized tools to support a wide range of these activities, possibly including:

1) Requirements Specification
2) Overall System Design
3) Program Design
4) Program Verification
5) Project Management

Clearly, the MAPSE does not emphasize any particular development methodology at the expense of any other. However a comprehensive APSE may encourage, or even enforce, one specific system development methodology.

4.3 DISCUSSION

The previous two sections have shown that (1) the SDL and MML languages have no dependency on CHILL (and vice versa) and (2) the APSE supports the incorporation of external tools such as SDL and MML. The significance of these two points will be discussed further herein.

The relationship that exists between SDL, MML, and CHILL poses no known portability problems. They exist in the same environment because of the fact that they support the same application. It is perhaps misleading to consider them as part of a "programming environment". One does not generate code for the other, for example. Rather, their interrelationship is more along the lines of "peaceful co-existence." SDL and MML existing in the same environment is somewhat analogous to a word processing package and an accounting package co-existing with a FORTRAN compiler. The accounting package may even be written in FORTRAN. But this situation does not create a dependency in a sense that restricts portability or replacement. Hence there is nothing to prevent another language (for example, Ada) from replacing CHILL in a particular environment which happens also to include SDL and MML. However, this view is directed towards a development type environment.

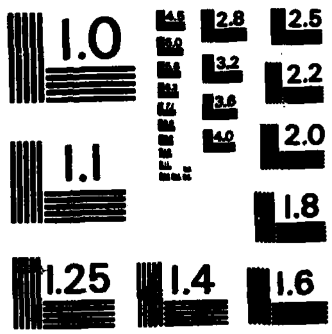
The relationship that exists within CHI[illegible]
and MML in a production environment is even easie[illegible]
consider. SDL and MML are basically off-line [illegible]
no requirement for them to co-exist with rea[illegible]
Hence, this places no additional restri[illegible]
context of replacing CHILL.

The above discussi[illegible]
whether Ada can replace CHILL [illegible]
part of a programming envir[illegible]
further area to conside[illegible]
fit within the Ada Pr[illegible]

The A[illegible]
programming env[illegible]
implementat[illegible]

AD-A121 938 EVALUATION OF ADA AS A COMMUNICATIONS PROGRAMMING LANGUAGE(U) SYSCON CORP SAN DIEGO CA A L BRINTZENHOFF ET AL. 31 MAR 81 DCA100-80-C-0037 3/4

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

provides more than adequate development support of Ada programs. The third level is the area of most interest to this study. This level provides the capability of extending the MAPSE to allow fuller support of particular applications or methodologies. In particular, this level can support the inclusion of the SDL/PR and MML tools. In fact, there is nothing to prevent the inclusion of the CHILL compiler and its associated tools as well. This is considered to be an attractive alternative to the question of Ada replacing CHILL in its environment.

SECTION 5
CONCLUSIONS

This section will present the overall conclusions of the Ada/CHILL comparative analysis effort. But first it is instructive to examine the conclusions of three other recent reports which deal with the same subject.

The first report was written in March 1980 by Mr. Kristen Rekdal (a member of CCITT study group XI and principal author of the CHILL language introduction). This report makes the claim that CHILL and Ada are technically very similar and that the differences are primarily political. In fact, the following paragraphs are significant:

> "These two languages have been designed with basically the same requirements in mind. The result is languages that are very similar both in power, structure and style. It is almost possible to do a detailed feature by feature comparison.
>
> From a purely technical point of view, there is reason to believe that either language could cover all purposes equally well. One may attempt to argue the preferability of details in one language above the other. Such comparisons will, however, be highly subjective and probably largely irrelevant.
>
> Language design is an exercise in making compromises, and some will be less pleasing than others. It is not difficult to find, in any language, properties to disagree with. But what counts is the overall result. There exists today no means by which it is

possible to detect any significant difference in overall language power or programmer productivity between so closely similar languages." /REKD80/

Mr. Rekdal then goes on to point out the political issues behind Ada and CHILL and concludes that "World agreement has been reached that, for right or wrong, CHILL is the language needed for SPC-programming." /REKD80/ He then advocates a "you go your way and I'll go mine" philosophy between CCITT/CHILL and DoD/Ada.

The second report details a study conducted under the auspices of the GTE Software Steering Committee by members of the Special Interest Group on High Level Languages /KORNXX/. This report elected to concentrate the comparative analysis of the two languages into four areas considered to be relatively new concepts in programming languages; modularization, data abstraction, parallelism, and exception handling. The authors uncovered no significant differences in the languages and concluded that

> "In the development of the four concepts, both Ada and CHILL have extended the features and facilities defined in basic PASCAL. Ada does not conflict with the CCITT requirements and is the most encompassing and ambitious effort we have encountered. Implementation of the complete language will be difficult and we believe that compilers supporting only subsets of the language will be available in the immediate future.
>
> CHILL more than adequately meets its design goals by supporting the development of real-time telecommunications software. It does not allow for any file handling or provide a 'real' number capability.

Most telecommunications software designers will therefore have to resort to supplemental languages to fulfill any requirements for file handling or scientific computations."

The third report was written by Mr. R.T. Boute of Bell Telephone Mfg. in Antwerpen, Belgium. /BOUT79/ Mr. Boute's report is by far the most detailed. His discussion of the two languages also centers around four issues: types, data abstraction, concurrency, and exception handling. Mr. Boute considers these features germaine to a communications oriented programming language and concludes that "Although no comparison was originally intended, Ada turns out to be definitely superior in the last three topics mentioned, as well as in the overall design." /BOUT79/ Furthermore, he goes on to say "The potential user is entitled to question the need for both languages, with all support and standardization problems it entails. The fast progress of Ada and the wide attention it has recently been getting may well establish its position before CHILL reaches Ada's present level of definition. In this case, and unless the CHILL design team decides on an approach which is superior to Ada in all respects, a second language would be superfluous." /BOUT79/ This again implies, as in the previous two reports, that the languages are so similar that the other is "superfluous."

The purpose of discussing these reports is to point out that studies performed by three diverse individual activities have generally arrived at the same conclusion, i.e. the technical similarity of Ada and CHILL.

Unfortunately, none of these reports addressed the entire spectrum of Ada and CHILL features and issues. The first report did not provide (and did not claim to provide) any technical justification on which to base its conclusions. The report was very clear in pointing out that the Ada/CHILL differences involved political rather than technical questions. The second report provided some limited technical information but isolated four areas (modularization, data

abstraction, parallelism, and exception handling) for purposes of comparison. This concentration was intentional and was performed to highlight what the authors termed "new concepts in programming languages." The third report also concentrated the comparison in four areas (types, data abstraction, concurrency, and exception handling) and provided detailed technical information in these areas, omitting only the purely sequential control structures and basic details of the languages considered trivial for comparison purposes.

Discounting the last two referenced reports for the stated reasons is not meant to be a negative judgment of their worth. Rather it is meant to point out that failure to explicitly cover all aspects of Ada and CHILL during a comparative analysis could perhaps result in a potential reader being misled into thinking either (1) the languages are identical in all unstated areas or (2) significant differences in these unstated areas are not being addressed. Also, it should be noted that all three reports dealt with preliminary Ada, not the recognized version of Ada defined in /USDO80b/.

For these reasons, the comparative analysis described within this report attempted to take a more global comparison approach which can be accepted in both technical and logical terms. The reader will recall that there were two basic questions to be answered during this study and they are reiterated here:

1) Can Ada be used as a direct substitute for CHILL in the context of CHILL being a programming language designed for circuit switching applications?
2) Can Ada be used as a direct substitute for CHILL in the context of CHILL being part of a programming environment containing CHILL, SDL, and MML?

The most common method of answering the first question would be to address the functional requirements of circuit switching applications and attempt to show that Ada meets those requirements. A more direct and less subjective

method is simply to follow deductive logic. For example, no one will argue the point that CHILL is a suitable language for circuit switching applications. Thus, if one wants to evaluate whether another language is also suitable, simply compare the features of this other language with CHILL. In Section 3 of this report, that feature by feature comparison was presented. This comparison showed that the languages are in fact nearly identical. Granted, there are minor differences (features exist in CHILL, but not in Ada, and vice versa), but the fact remains that the differences are virtually insignificant when considered in totality. One can therefore conclude that Ada can be used as a direct substitute for CHILL in the context of CHILL being a programming language designed for circuit switching applications.

To answer the second question a further argument must be proposed. In Section 4 the CHILL/SDL/MML environment was examined. It was shown that no critical dependency exists between these three entities. In particular, the SDL and MML tools exhibit no characteristics which force them to depend on CHILL (or vice versa). Thus, in answer to the second question there is nothing to prevent Ada from coexisting with SDL and MML in a particular programming environment. However, we demonstrated in Section 4 that a more complete and useful capability can be formed by using the Ada Programming Support Environment (APSE). The APSE, as currently defined within the STONEMAN document can support the incorporation of external tools at its "outermost" level. Therefore, a very powerful support environment for SPC switching system applications can be formed by the incorporation of SDL and MML into the APSE. In fact, there is nothing to prevent the CHILL capability from being incorporated as well, allowing Ada programs to coexist and interface with CHILL programs, where appropriate. This is seen as a powerful, logical approach to the Ada/CHILL duality and it is a solution that the CHILL proponents can neither offer nor argue against.

In addition to the above, two other points are relevant to this discussion.

Both DoD and CCITT have stated their desire for defining a language standard for their respective application areas. Having a programming language achieve a standard level is advantageous to many activities, not the least of which might be configuration management, quality control, documentation, and training. Allowing (or not strictly controlling) the proliferation of compiler subsets tends to defeat the purpose of establishing a language standard. All too frequently, there is incompatibility among the subsets. Occasionally, the subset fails to accurately reflect the standard from which it is supposed to have been derived.

DoD is seeking to prevent this condition from occurring. They are doing this by forbidding the recognition of Ada compiler subsets within their application domain. Every Ada compiler will be required to recognize every legitimate Ada statement. This obviously does not prevent independent compiler development outside their domain, but at least it restricts the proliferation of subsets within their own environment. Additionally, no development activity will be able to call a subset compiler an Ada compiler because of the copyright restrictions which DoD intends to place on the use of the name.

CCITT, on the other hand, has not yet been able to establish firm control over the generation of CHILL compiler subsets within their own sphere of influence. This is evidenced by several ongoing trial compiler development activities. Whether these compilers are faithful subsets of the CHILL definition or are, in fact, compatible with each other is unknown at this time. The point is that the CHILL designers/proponents have to date failed to adequately control this condition, and this is considered contradictory to their stated goals.

Another closely related question then is: How does one validate these compiler subsets or compilers which have been generated within different development environments? The answer is that the defining authority must require that compiler validation be performed. Furthermore, the defining

activity or, at least, the implementing activity must establish procedures to be used to certify that the compiler in question meets the established language standard.

The STEELMAN document states: "There will be a standard definition of the language. Procedures will be established for standards control and for certification that translators meet the standard." /USDO78/ The DoD obviously intends to keep Ada under tight configuration control and to ensure that compilers do not introduce dialects through inconsistent implementations. In particular, a language control agent (which includes a compiler validation facility) is required to be in place before DoD will accept a language for the approved list, as stated in DoDI 5000.31. Toward this end, a contract has been awarded by DoD to develop an initial Ada Compiler Validation Capability (ACVC) to be available by late 1980 and a complete state of the art capability by late 1981.

CCITT, on the other hand, has not yet been able to establish a firm commitment to a compiler validation facility. And, due to the fact that several trial compilers are in the late stages of development, it appears unlikely that satisfactory validation efforts will be possible. Again, this seems contrary to CCITT goals.

In summary, it is felt that a strong case has been presented for Ada being used as a programming language for circuit switching applications. It has been shown that Ada is equal or superior to CHILL in almost all aspects ranging from availability and definition of language features to strict control over compiler dialects. Moreover, the study has produced no evidence which precludes Ada from being used in other, more general, telecommunications programming applications, as well. Many people believe that Ada could emerge as the universal programming language standard by the end of the decade, and therefore, there appears to be no reason why the communications community should not take advantage of Ada's power and appeal in all of their present and future software development activities.

(This Page Intentionally Left Blank)

APPENDIX A
REFERENCES

/BOUT79/ Boute, R.T., Ada and CHILL: A joint language evaluation. Report RTB-7908. Bell Telephone Mfg. Cy., Antwerpen, August 1978.

/BRIN73/ Brinch Hansen, P., Operating System Principles. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

/BRIN78/ Brinch Hansen, P., Distributed Processes: A concurrent programming concept. Comm. ACM Volume 21, Number 11, November 1978, pp 934-941.

/CCIT78/ CCITT, Report on the Meeting held in Geneva from 19 to 23 June 1978. COM XI-No 200-E, June 1978.

/CCIT80a/ CCITT, Draft Recommendation Z.200: Proposal for a Recommendation for a CCITT High Level Programming Language (CHILL). February, 1980.

/CCIT80b/ CCITT, Proposed Revised and Expanded Recommendations for the CCITT Specification and Description Language (SDL). AP VII-No. 20-E, June 1980.

/CCIT80c/ CCITT, Proposed New and Revised Recommendations for the CCITT Man-Machine Language (MML). AP VII-No. 22-E, June, 1980.

/HOAR78/ Hoare, C.A.R., Communicating Sequential Processes. ACM 21, 8 (August, 1978), 666-677.

/ICHB79a/ Ichbiah, J.D., et al., Preliminary Ada Reference Manual. ACM SIGPLAN Notices 14, 6 (June, 1979), Part A.

/ICHB79b/ Ichbiah, J.D., et al., Rationale for the design of the Ada Programming Language. ACM SIGPLAN Notices 14, 6 (June, 1979), Part B.

/KORNXX/ Kornfeld, C., et al., "Development of New Language Concepts in Ada and CHILL," GTE Software Steering Committee, undated.

/NCSY80/ National Communications System Introduction to the CCITT High Level Language, NCS TIB 80-1, January, 1980.

/REKD80/ Rekdal, Kristen, CHILL, Ada, and ESL, (Technical Report), March, 1980.

/USDO78/ U.S. Dept. of Defense, STEELMAN Requirements for High Order Computer Programming Languages. June, 1978.

/USDO80a/ U.S. Dept. of Defense, STONEMAN Requirements for Ada Programming Support Environments. February, 1980.

/USDO80b/ U.S. Dept. of Defense, Reference Manual for the Ada Programming Language. July, 1980.

(This Page Intentionally Left Blank)

PLAN FOR

EVALUATION OF ADA

AS A

COMMUNICATIONS AND TRUSTED SOFTWARE

PROGRAMMING LANGUAGE

(This Page Intentionally Left Blank)

PLAN FOR
EVALUATION OF ADA
AS A
COMMUNICATIONS AND TRUSTED SOFTWARE
PROGRAMMING LANGUAGE

ABSTRACT

The availability of the new programming language, Ada, presents new opportunities for developing quality software through the use of language features used previously only in research environments. With the new features, however, new controls in the form of programming standards and guidelines will be required to assure that the potential for producing quality software is actually achieved. As a means of formulating these standards and guidelines, Ada will be used to implement, on a prototype basis, a communications application which consists of the AUTODIN II Segment Interface Protocol/Advanced Data Communications Control Procedure (SIP/ADCCP) and a trusted software application which consists of the Advanced Command and Control Architectural Testbed (ACCAT) GUARD software. This Evaluation Plan establishes the approaches to be used in designing, developing, and testing the software, evaluating the efficiency and effectiveness of Ada as used in these applications, and identifying standards and guidelines to assure overall software quality in the use of Ada.

(This Page Intentionally Left Blank)

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

LIST OF ILLUSTRATIONS

EXECUTIVE SUMMARY

The availability of the new programming language, Ada, presents new opportunities for developing quality software through the use of language features available previously only in research or small-scale software development environments. Although the existing, July 1980 version of Ada has resulted from extensive, open review, test and evaluation by individuals from government, industry, and educational institutions, to date no major software design or development effort using Ada as the implementing language has been undertaken.

Based on limited actual use of Ada for implementations of stand-alone applications, preliminary results indicate that different software development approaches may be required to effect the optimal use of Ada. These include, for example, the use of Ada as a software design language as well as the implementing language, changes in the approach to modularization including the definition of compilation units, additional emphasis on the use of the data abstraction capabilities, and the use of the Ada tasking constructs for designing and implementing concurrent programming applications. Another separate, but not unrelated, issue is the effect of individual programming styles on the production of quality software, particularly with regard to maintainability of software. Ada is a rich, powerful, and versatile language which provides the creative programmer with many opportunities and among these is also the opportunity for misuse or abuse of the language features. Finally, another area of concern is how suitable, effective, and efficient the features of Ada are with regard to specific classes of applications.

As a means of evaluating Ada in the above context, the Defense Communication Agency, through the Defense Communication Engineering Center, has selected two classes of software to be implemented using Ada. The first is a communication application, which is the Segment Interface

Protocol and Advanced Data Communication Control Procedure (SIP/ADCCP) used in the AUTODIN II system. The second is a trusted software application, the Advanced Command and Control Architectural Testbed (ACCAT) GUARD, which functions as a trusted process for permitting the controlled exchange of information between separate SECRET and TOP SECRET systems. This Evaluation Plan identifies the approaches, criteria, and key elements required to perform an evaluation of Ada in Phase II of this project with regard to its suitability, effectiveness, and efficiency in the SIP/ADCCP and ACCAT GUARD applications. As a result of the evaluation, a set of programming standards and guidelines will be defined to assure that the potential for producing quality software is actually achieved.

The methodology presented in the Evaluation Plan consists of defining the concept of software quality and establishing software quality factors related to software development and maintenance, and to software performance which provide the basis for the evaluation of Ada. These software quality factors are in turn related to more detailed criteria and the definition of software metrics to evaluate specific, quantitative aspects of the developed application software. In addition, specific, application-oriented language features which will be used to evaluate Ada as a suitable programming language are also defined.

The software development will be organized as a mini software development project with nominal standards, internal reviews, milestones, and semi-formal testing of the developed software. The objective is to emulate, to the maximum extent practicable, the phases and operations of a major software development effort to assure that results obtained will not be out of context when applied to such efforts. By and large, the two applications will be treated as separate and distinct development efforts in order to obtain as much diverse experience and knowledge as possible regarding the suitability of Ada. The execption to this will be a small amount of programming by each programmer in the other

application area to help in assessing maintainability issues and developing broader perspectives regarding the best use of Ada.

The evaluation will comprise the acquisition and analysis of data from three sources. These are error statistics (compile-time and run-time), software structure analysis (modularity, internal structure, assessment of Ada features), and programmer interviews (overall qualitative evaluation, identification of problem areas, design rationale). The results of the data analysis will be used to identify specific or generic problems which were encountered and to formulate solutions in the form of standards and guidelines which will diminish or eliminate those problems.

The software development tools which will be used consist of the Ada/ED translator-interpreter which has been developed by the Courant Institute of Mathematical Sciences of New York University under the auspices of the U.S. Army Communication Research and Development Command (CORADCOM) and standard Digital Equipment Corporation VAX 11/780 system software. Plans include the hosting of Ada/ED on the VAX 11/780 at the University of California at San Diego Computer Center for the development and evaluation effort. The developed software and Ada/ED will subsequently be delivered to DCEC for operation on its VAX 11/780.

The planned development and evaluation effort spans a period of approximately thirteen months and will utilize the skills of two senior system analysts and a project manager who will also have major responsibilities in the evaluation effort.

(This Page Intentionally Left Blank)

SECTION 1
INTRODUCTION

1.1 PURPOSE

The purpose of this Evaluation Plan is to identify all the key elements which will be required to evaluate the suitability of Ada as a language for developing communications and trusted software. These key elements include the levels of testing and evaluation to be performed, the specific requirements and approach for each level, the responsibilities of all personnel associated with the requirements, identification of test site, hardware and software, the evaluation schedule and relevant quality assurance factors.

The goals associated with the implementation of this Evaluation Plan are twofold: the first is to assess the ability to develop quality communication and trusted software using Ada as the programming language; the second is to provide a set of guidelines and standards, which, if implemented, will help to assure the development of quality software using Ada.

1.2 SCOPE

The scope of the Evaluation Plan will encompass two areas of quality in communications and trusted software which are development and performance. The development area will be concerned with assessing software quality factors related to the development, maintenance, and modification of software. These factors include, for example, testability, flexibility, and maintainability. The performance area will be concerned with assessing software quality factors related to the run-time performance of the software. These factors include, for example, reliability, correctness, and efficiency.

Two separate applications will be implemented in order to evaluate Ada with regard to the development and performance quality factors. The communications applications involves the implementation of the AUTODIN II Segment Interface

Protocol (SIP) and the AUTODIN II Advanced Data Communications Control Procedures (ADCCP); the other application, related to computer security, is the Advanced Command and Control Architectural Test Bed (ACCAT) GUARD function which is an adjunct to the Kernelized Secure Operating System (KSOS).

In order to have the Ada evaluation produce results which are relevant to real-world software development, the Phase II evaluation will be structured as a mini-software development project. The project phases will consist of macroscopic and microscopic design phases (using the present top-level software designs), code/debug/modify, test plan, procedure and test data development, software testing, and software operation via simulation of inputs. As the project progresses through the various phases, data which are related to the software quality factors will be collected and analyzed to evaluate Ada and to formulate the guidelines and standards.

1.3 SCHEDULE SUMMARY

The detailed schedule for the proposed test and evaluation effort is presented in Section 8, Schedule. The schedule, as proposed, spans a period of thirteen months. A brief description of each of the task categories, as shown in Figure 1.3-1, is given below along with the corresponding approximate time periods. An Ada Orientation task of one month will be devoted to establishing initial guidelines for the use of Ada, acquiring the Ada translator-interpreter and providing indoctrination on the concepts embodied in the Ada language constructs. The Software Design task, encompassing approximately five months, will provide the macroscopic and microscopic designs and the development of test plans, procedures and data. The Code/Debug/Modify task, encompassing approximately four months, will be concurrent, in part, with the Integration/Test task. These tasks will result in the implementation of the ACCAT GUARD, SIP/ADCCP, test support software and the testing of the software. The Evaluation Procedures Development task, encompassing approximately five months, will produce detailed procedures for acquiring data

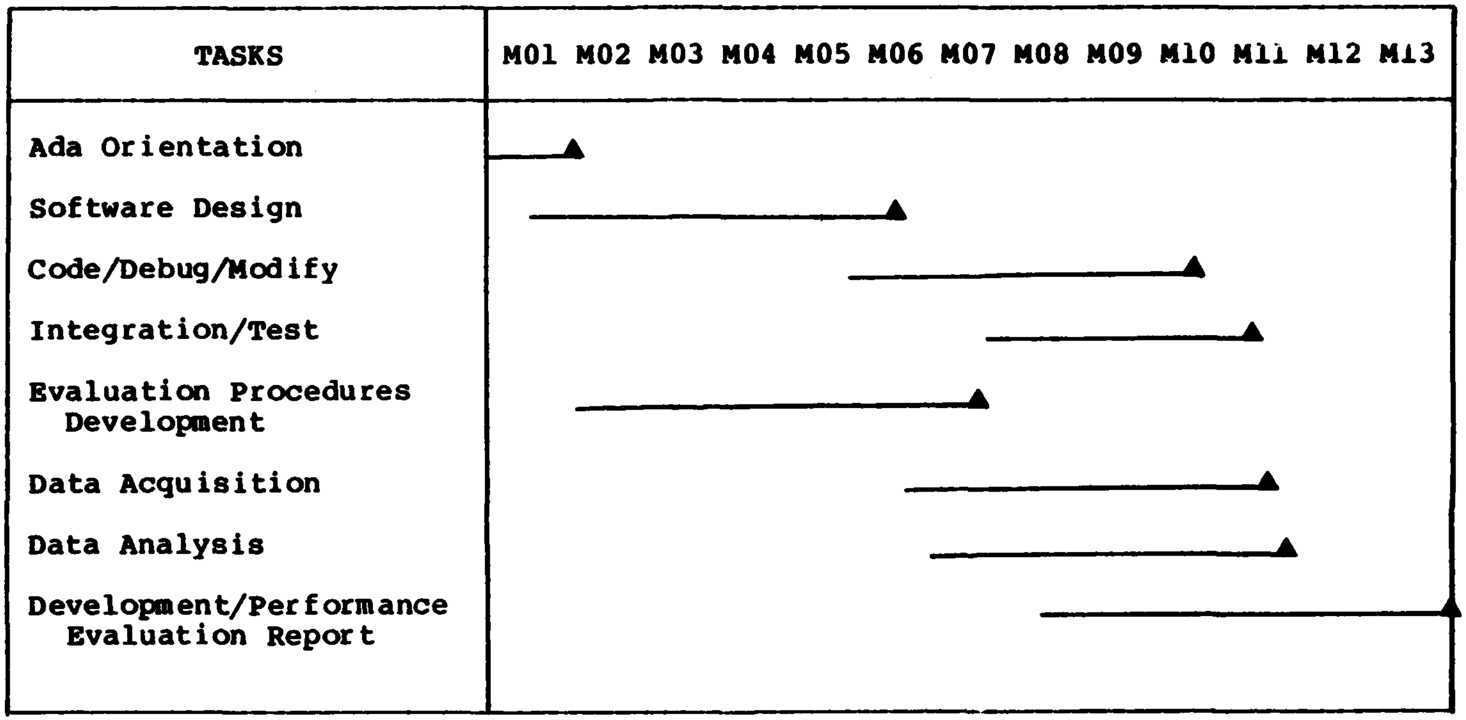


Figure 1.3-1. Summary Schedule

during the Data Acquisition Task. The Data Acquisition Task, encompassing approximately five months will run concurrently, in part, with the Data Analysis task. These tasks will result in the collection and analysis of error statistics, software statistics and results of programmer interviews. The Development/Performance Evaluation Report task, encompassing approximately five months, will produce the draft and final versions of the Development/Performance Evaluation Report and provide a summary oral presentation based on the draft report.

1.4 TEST AND EVALUATION LEVEL SUMMARY

There will be a total of four test and evaluation levels. The two test levels will comprise module and system integration testing. The two evaluation levels will comprise software-development evaluation and software-performance evaluation.

1.4.1 Test Level Summary

The testing of the two test levels will be designed to assure that the software of each application meets its respective specifications established in the requirements and design documentation irrespective of the language, standards and guidelines, programming style, and similar characteristics associated with the development process. This testing, as the names of the test levels imply, will be performed as the software progresses through its development phases. Therefore, the development of test plans, procedures, specifications, test data and the conduct of the testing will be defined and implemented as part of the mini software development effort and reference to them in the Evaluation Plan will be only cursory.

1.4.2 Evaluation Level Summary

The two evaluation levels, software development and software performance, will be designed to measure software development and software performance with regard to the fact that Ada is the implementing language. Thus, this Evaluation

Plan will focus on how the software development and performance will be measured, what the measurement criteria are and how they will be used to assess the suitability of Ada for developing communication and trusted software.

The objective of the software-development evaluation will be to determine what problems, if any, result from the use of Ada as the implementation language and to then formulate suitable guidelines or standards which will eliminate or reduce the problems. The evaluation methods will use quantitative data, such as the number and type of errors encountered during the compilation and testing process and the size of the programs, and qualitative data, such as programming styles, software complexity, and software organization.

The objective of the software-performance evaluation will be to determine how well the Ada constructs, in their machine implementation, perform during the execution of the software. At present, it appears that a software performance evaluation will be limited for two reasons. First, there will be no production quality compiler available during the planned Phase II period. Second, the Ada translator-interpreter being produced by New York University is believed to be too far removed from the planned production quality compilers to permit any meaningful extrapolation of performance results. However, as indicated below, certain software performance factors can still be evaluated. In fact, with the Ada translator-interpreter, the only software performance factor which cannot be evaluated is performance efficiency which deals with such factors as execution and memory efficiencies and optimizations.

(This Page Intentionally Left Blank)

SECTION 2
APPLICABLE DOCUMENTS

2.1 MILITARY STANDARDS AND SPECIFICATIONS

a. /M16778/

Department of Navy, Military Standard, Weapon System Software Development; MIL-STD-1679 (Navy), 1 December 1978.

b. /D21478/

Department of Navy, Data Item Description, Computer Program Test Plan; DI-T-2142, 29 November 1978.

c. /M84773/

Military Standard - Format Requirements for Scientific and Technical Reports Prepared By or For the Department of Defense; MIL-STD-847A, 31 January 1973 including Update Notices 1, 2.

2.2 SYSTEM SPECIFICATIONS AND REFERENCES

a. /WOOD78/

J.P.L. Woodward, "ACCAT GUARD System Specification (Type A)", MTR-3634, The MITRE Corporation, Bedford, MA, August, 1978.

b. /LOG179a/

LOGICON, "Formal Specification of GUARD Trusted Software (Draft)," ARPA-78C032303, September, 1979.

c. /LOG179b/

LOGICON, "ACCAT GUARD Program Development Specification (Type B5)," ARPA-78C0323-01, February, 1979.

d. /BALD79/

David L. Baldauf, "ACCAT GUARD Overview," the MITRE Corporation, Bedford, MA, November, 1979.

e. /WEST79/
Western Union, "Initial AUTODIN II Segment Interface Protocol (SIP) Specification," (System Engineering Technical Note TN 78-07-31), DCA 200-C-637-P003, 5 March 1979.

f. /WEST78/
Western Union, "AUTODIN II Design Executive Summary," Western Union Telegraph Company, McLean, Virginia 22101, 18 May 1978.

2.3 OTHER GOVERNMENT REFERENCES

a. /USDO80a/
United States Department of Defense, "Reference Manual for the Ada Programming Language," United States Government, Director of Defense Advanced Research Projects Agencies, July, 1980.

b. /USDO80b/
United States Department of Defense, "Requirements for Ada Programming Support Environments," "Stoneman," United States Government, February, 1980.

2.4 NON-GOVERNMENT REFERENCES

a. /BBNI76/
Bolt, Bernek, and Newman, Inc., "Development of a Communications Oriented Language, Parts I and II", Report No. 3261, 20 March 1976.

b. /SRII78/
SRI International, "Verification of Communications-Oriented Language Programs," SRI International Final Report, Project 6413, August, 1978.

c. /HALS77/
Maurice H. Halstead, Elements of Software Science, Elsevier North Holland, Inc., New York, 1977.

d. /COOP79/

John D. Cooper and Matthew J. Fisher Editors;
Software Quality Management
Petrocelli Books, Inc., New York, 1979;
"An Introduction to Software Quality Metrics"
by James A. McCall.

(This Page Intentionally Left Blank)

SECTION 3
TEST AND EVALUATION REQUIREMENTS

3.1 OVERVIEW

The following sections establish the basic requirements for the Ada evaluation. Section 3.2 identifies the overall approach to conducting the testing of the software and the evaluation of Ada. Section 3.3 establishes the concept of software quality and identifies specific elements of software quality that will be evaluated in the context of using Ada as the implementation language. Section 3.4 identifies the software development methodology within which the application software will be developed. Section 3.5 presents the approach to collecting and analyzing the data which will be used in evaluating Ada. Section 3.6 identifies how results and conclusions of the Phase II effort will be organized and presented.

3.2 SOFTWARE TEST AND EVALUATION APPROACH

This section identifies the overall approach which will be taken with respect to the testing of the developed applications and the evaluation of Ada as a suitable programming language for the selected applications.

3.2.1 Software Evaluation Approach

The overall approach to evaluating Ada as a programming language suitable for developing communication and trusted software will be to develop those types of software and measure the extent to which Ada is adequate by evaluating the developed software. In order to accomplish this, two critical elements must be defined. These elements are the concept and supporting details of software quality and the software development methodology which will be used to produce the software.

The software development methodology will provide a framework within which the application software will be developed. The primary objective of this is to emulate, as closely as possible, the salient aspects of a major software development effort. This will assure that information obtained during the evaluation and subsequent conclusions will be germane to similar software applications developed under actual project conditions.

The software quality concepts and supporting details are the second critical element since they will be the basis for determining what evaluation criteria are to be used. The quality concepts fall into two broad areas which are software quality factors and application-oriented requirements. The software quality factors will be used to define the constituents of software quality at the conceptual level. These in turn will be related to lower-level entities which can be either measured quantitatively or evaluated qualitatively to determine how well Ada supports the factors and to assess the influence of individual programming styles. The application-oriented requirements will provide a basis for evaluating Ada with respect to the suitability of Ada constructs for addressing data design and control structures which are frequently found in the types of software being developed.

Finally, by acquiring the necessary data as the software development progresses and analyzing the data during and at the conclusion of the development, an assessment of Ada will be made. The results of this assessment will then be translated into a set of programming standards and guidelines to assist in the development of quality software. Moreover, even if Ada is found highly suitable, there will still be the need for developing and maintaining programming guidelines and standards to deal with issues such as programming styles which transcend the features of any language.

3.2.2 Software Testing Approach

The approach to testing the SIP/ADCCP and ACCAT GUARD application will consist of exercising the software via the use of specially designed test-support software which is identified in Section 7.

The objectives of the software testing in each application area will be twofold. First, the standard test objective of discovering problems and correcting them will be employed to produce applications which satisfy their requirements. An additional test objective, however, will also be defined. Since the ultimate goal of this evaluation is to formulate a set of standards or guidelines for communications and trusted software programs written in Ada, the results of testing will serve as input data for conducting the Ada software development and software performance evaluations. Specifically, the errors detected will be analyzed and organized into classes or groups in order to determine if there are broad classes of problem areas in understanding or using the Ada constructs which warrant the definition of specific guidelines or standards.

3.2.3 Application Overviews

In order to provide a more complete understanding of the SIP/ADCCP and ACCAT GUARD applications and to provide the proper context for the test and evaluation, an overview of each application is given below. For more detailed information, the references of Section 2 may be used.

3.2.3.1 SIP/ADCCP Overview

The SIP/ADCCP applications represent the two lowest-level protocol layers of the AUTODIN II packet-switching network. The functional location of the protocols is shown in Figure 3.2-1.

The SIP is designed to accept data, commands, and responses from the next higher AUTODIN II protocol layer, the Transmission Control Protocol (TCP), process them accordingly and effect the transfer of packets via the Packet Switch Nodes

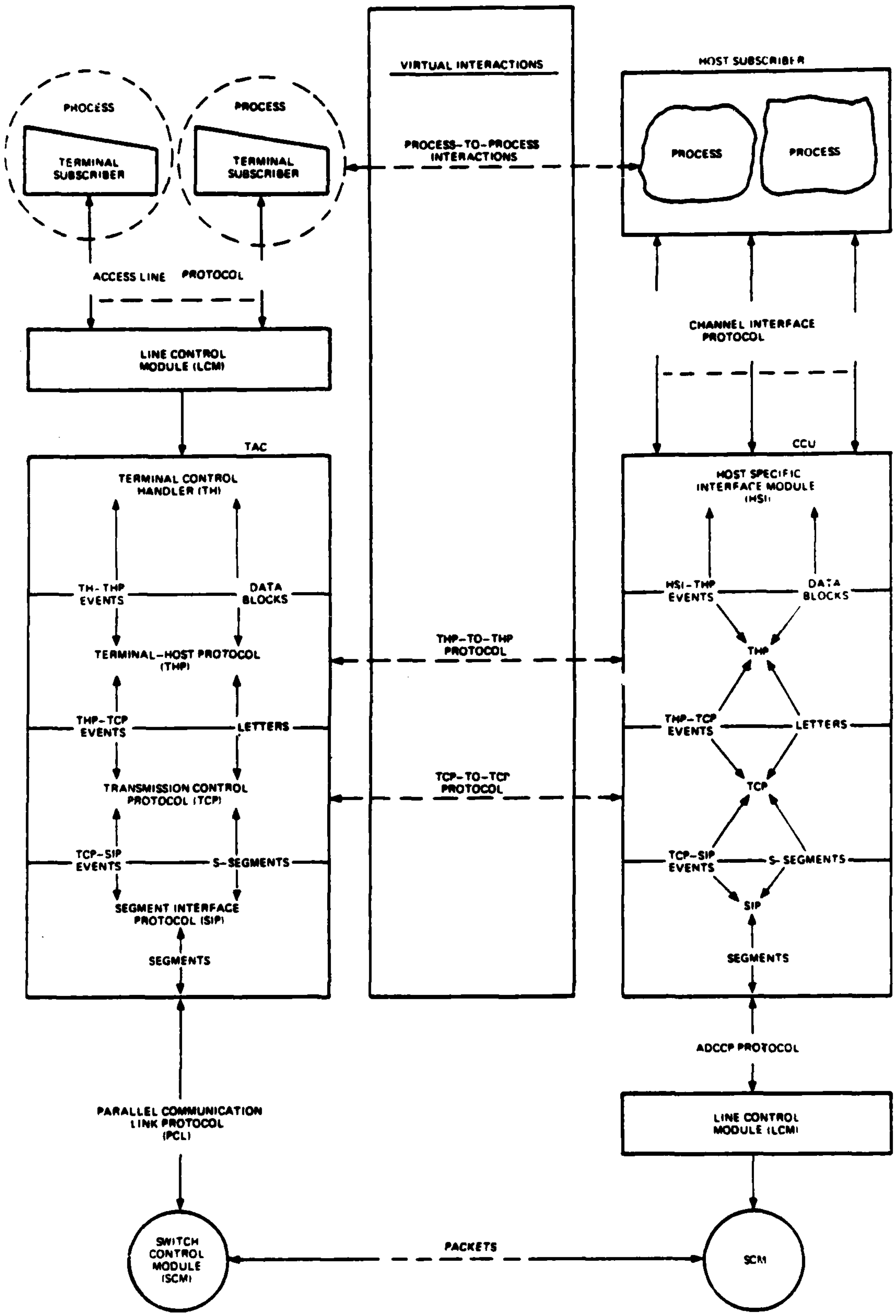


Figure 3.2-1. Levels of Process-to-Process Protocols (Reprint: Figure EC.2-1 /WEST78/.)

of the AUTODIN II network for transmit operations. For receive operations, the SIP is designed to accept data, commands, and responses from the Packet Switch Nodes, process them accordingly, and effect the transfer of packets to the TCP.

To control the transmission of packets (data, commands, and responses) on an inter-PSN basis using the Mode VI access lines, the SIP will use the Advanced Data Communication Control Procedure (ADCCP). Thus, the ADCCP functions as the line control protocol, the protocol layer which is lowest and next to the hardware, of the AUTODIN II network. In particular, the ADCCP will be used to control Mode VI line access for synchronous character and synchronous binary data transmissions.

3.2.3.2 ACCAT Guard Overview

The ACCAT GUARD application has been designed to provide secure, monitored, controlled transfer of data between a high-level (TOP SECRET) and a low-level (SECRET) system. An overview of the system configuration is given in Figure 3.2-2. Separation of high-level and low-level entities (files, queues) is maintained by use of the Kernelized Secure Operating System (KSOS). To accomplish the intersystem transfer of data, the high-level and low-level software in the ACCAT GUARD system is interfaced by two trusted processes. The Upgrade Trusted Process (UGTP) is reponsible for transferring low-level information to the high-level system; the Downgrade Trusted Process (DGTP) is responsible for transferring high-level information to the low-level system under the control of Sanitization Personnel (SP) and a Security Watch Officer (SWO). The SWO is responsible for downgrading information and providing controlled and monitored transfer of data to the low-level system. The SP are responsible for sanitizing information which is deemed unsuitable for downgrading by the SWO in its present form. Six types of intersystem transfers and two types of operations can be performed. The transfers consist of high-low and low-high mail and high-low queries and responses and low-high queries and responses. The queries are

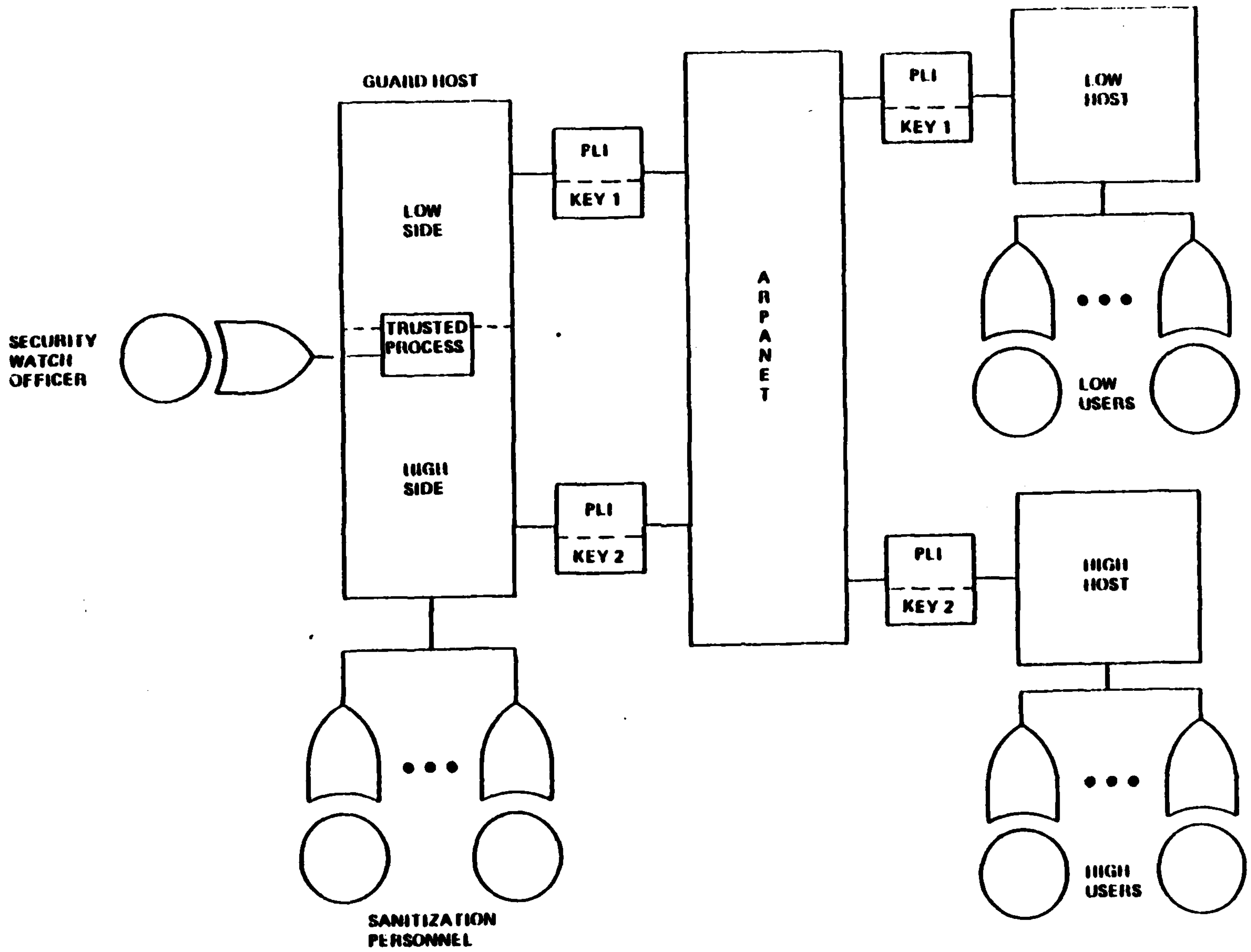


Figure 3.2-2. ACCAT GUARD Configuration (Reprint of Figure 1 /LOGI796/)

in either canonical form (preformatted data base queries) or English-language form which is then translated into canonical form by the SP. The two types of operations are sanitization which entails selecting, reading, and editing text files to remove TOP SECRET information and downgrading which entails enforced reviewing of data to be transferred to the low system and either accepting or rejecting the transfer.

Although only two of the processes are trusted processes and thus subject to formal specification and verification, several other processes on both the high and low side are required to support the trusted processes. The processes and their interactions are shown in Figure 3.2-3.

3.2.4 Test and Evaluation Constraints

As indicated above, the objective of the evaluation is to assess Ada in both the software-development and software-performance areas. Within the time frame of the currently planned Phase II, however, there are certain limitations or constraints which exist and which impact the extent of the evaluations. The constraints and their impacts are summarized below.

First, the Ada language processor which is proposed for this effort is not mechanized as a production-quality compiler which would include such features as memory-space and execution-time optimization of the generated target code. Instead, the processor, Ada/ED, is mechanized as a translator-interpreter which is implemented in the high-level language, SETL. Thus, from a performance standpoint it will not be possible to assess the effect of memory space and execution time trade-offs or to extrapolate performance with regard to these criteria to other computer architectures or operating system types.

A second type of constraints will also need to be considered with respect to the ACCAT GUARD application. The existing ACCAT GUARD application is designed to operate under Western Electric UNIX (a trademark of Bell Laboratories), Version 6.0 and specifically takes advantage of those features

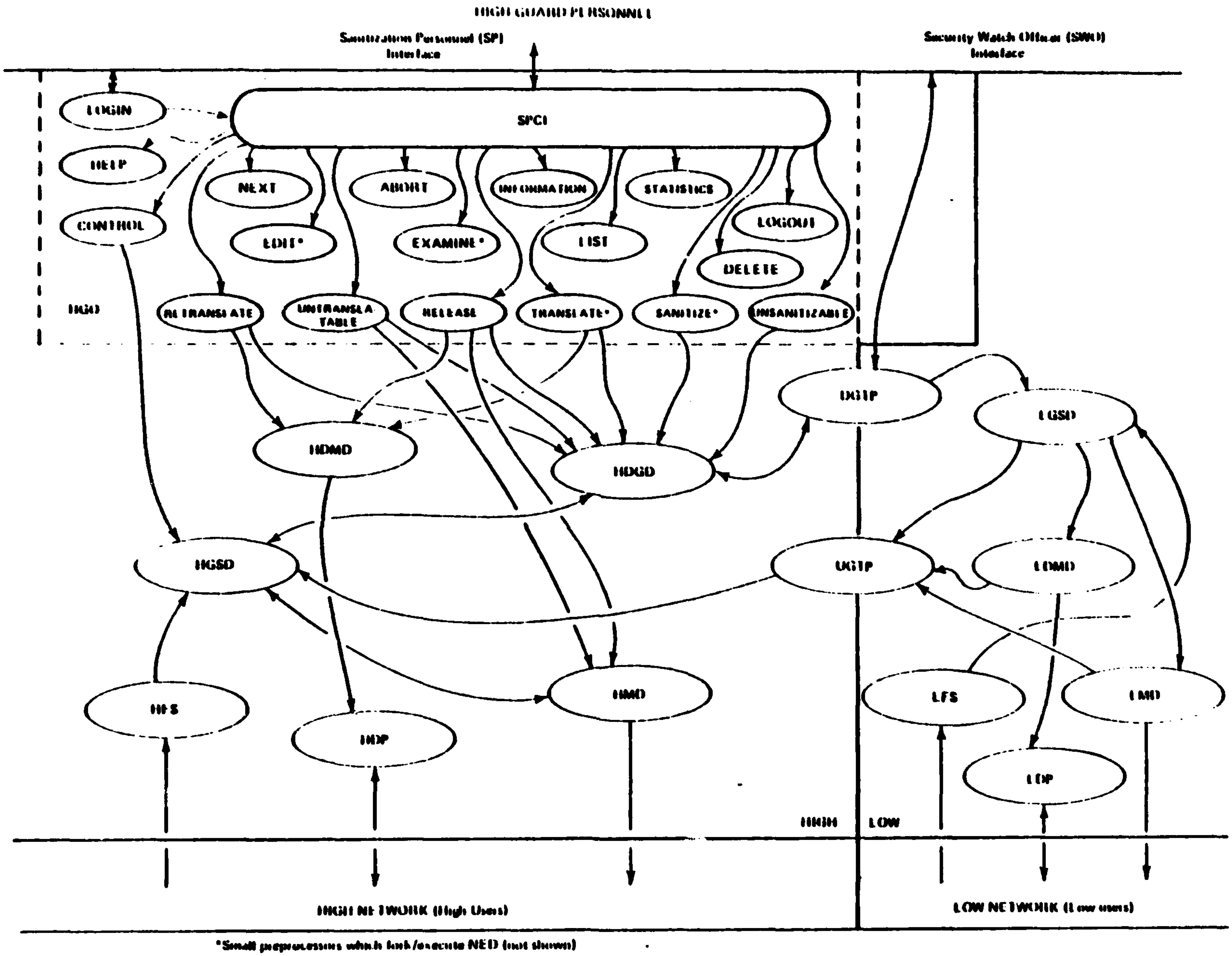


Figure 3.2-3. Basic ACCAT GUARD Process Information Flow
(Reprint of Figure 2 /LOGI796/)

such as ports, interprocess communication features and a text editor all of which operate under UNIX. To the maximum extent possible, the trusted software components will be isolated from these peripheral issues; however, there may be some additional limitations which need to be imposed to make the development tractable.

Finally, in terms of more completely addressing the performance evaluation, several true complilers are being developed and are projected for completion in the second half of calender year 1981. Three of these are Intel Corporation's implementation for the IAPX 432 computer, Telesoftware's implementation under UCSD Version 4 Pascal (host and target), Control Data Corporation's implementation on its CYBER class computer, and Intermetrics Inc.'s compiler for the DEC 20 with TOPS-20. If one or more of these compilers were available, then it would be possible to complete the evaluation as currently planned and rehost the developed software to one or more of the available compilers to specifically evaluate execution and memory efficiency.

3.3 SOFTWARE QUALITY

This section identifies the software parameters and application-oriented language requirements which affect the development and performance of quality software.

3.3.1 Software Quality Factors

In order to define software quality, a hierarchical set of software quality parameters will be defined. At the highest level is the concept of software quality which is "the composite of all attributes which describe the degree of excellence of computer software", /COOP79/. Next are the conceptual software quality factors which represent attributes that are desirable for software to have. At the lowest level are the measurable parameters which can be related to the software quality factors.

The software quality factors defined in this section consist of those which are related to and impact on software development and software performance as presented in /COOP79/. It should be noted that "the degree of excellence" required of software is not absolute since different organizations and projects may have different objectives. For example, "throw-away" code need be given very little consideration with respect to life-cycle maintainability. In addition, some software quality factors such as transportability and efficiency are potentially in conflict and thus necessitate a trade-off or balance to be struck.

3.3.1.1 Software Quality Factors (Development)

The software quality factors defined in this section are those which are related to or impact on the software development, maintenance, and modification process as opposed to software performance. Table 3.3-1 lists the software quality factors and their definitions. Their associated criteria are defined in Table 3.3-3.

3.3.1.2 Software Quality Factors (Performance)

The software quality factors defined in this section consist of those which are related to or impact on the performance of software implemented in Ada. Table 3.3-2 lists the software quality factors and their definitions. Their associated criteria are defined in Table 3.3-3.

3.3.2 Criteria for Software Quality Factors

The criteria identified in Table 3.3-3 represent a set of independent attributes which software may possess both with regard to software development and software performance. In many instances, an individual criterion will be correlated with more than one software quality factor. Because of this, the total set of criteria is presented here even though some criteria also support, either partially or exclusively, the software performance quality factors. The interrelationships between the software quality factors and the software quality

Table 3.3-1. Software Development Quality Factors

EFFICIENCY I

A measure of the extent to which algorithms are or can be represented in compact format using the available language constructs.

FLEXIBILITY

A measure of the extent to which an operational program can be modified to include new functional capabilities.

INTEROPERABILITY

A measure of the extent to which two operational programs of different systems can be coupled or interfaced without modification to enhance performance or functional capabilities.

MAINTAINABILITY

A measure of the extent to which an error in an operational program can be identified, isolated, and corrected.

REUSABILITY

A measure of the extent to which an operational program can be used as a component in another application without modification.

TESTABILITY

A measure of the extent to which a program can be readily tested to assure that performance criteria are met during the development, maintenance, and modification phases.

TRANSPORTABILITY

A measure of the extent to which an operational program can be readily transferred to a different hardware or software environment and perform correctly without modification.

Table 3.3-2. Software Performance Quality Factors

CORRECTNESS
A measure of the extent to which an operational program complies with its specifications, performs its functions and produces acceptable results.

EFFICIENCY II
A measure of the extent to which an operational program makes optimal use of system resources including CPU time, memory, and peripherals.

INTEGRITY
A measure of the extent to which an operational program performs only its intended functions and does not overtly or covertly perform any other functions.

RELIABILITY
A measure of the extent, with regard to frequency and criticality of failures, to which a program can be expected to perform its required functions in its intended environment.

ROBUSTNESS
A measure of the extent to which an operational program is able to acceptably manage or respond to conditions outside its intended operational environment.

USABILITY
A measure of the extent to which program users can prepare input data for, interpret output data from, and control operation of the program and learn to use the program in its intended environment.

Table 3.3-3. Criteria for Software Quality Factors
(Page 1 of 3)

ACCURACY
The attribute of software that provides for the usability of the computational results with regard to correctness, precision, and timeliness.

COMMUNICATIONS COMMONALITY
The attribute of software which provides for the use of standard protocols and mechanisms for the interfacing of two software components.

COMMUNICATIVENESS
The attribute of the software that provides outputs which can be readily assimilated by a user and requires inputs which can be readily supplied by the user.

COMPLETENESS
The attribute of software that provides for the full implementation of all functions and capabilities specified.

CONCISENESS
The attribute of software that provides for implementation of a function with the use of a minimum quantity of source code.

CONSISTENCY
The attribute of software that provides uniform design and implementation techniques, guidelines, standards, and notation.

DATA COMMONALITY
The attribute of software which provides for the use of standardized data formats and representations.

Table 3.3-3. Criteria for Software Quality Factors
(Page 2 of 3)

ERROR MANAGEMENT
The attribute of software to correctly detect, isolate, manage, and inform on all specified error conditions.

GENERALITY
The attribute of software that permits it to handle a broader scope of problems or conditions than those specified.

HARDWARE ARCHITECTURE COMPATIBILITY
The degree to which hardware elements and their configuration are effectively used by application programs.

HARDWARE INDEPENDENCE
The attribute of software that indicates the degree of coupling between the language constructs and the hardware on which the software will operate.

INSTRUMENTATION
The attribute of software which provides for the control or display of intermediate conditions, events, or data on a conditional or non-conditional basis.

LANGUAGE CONSTRUCTS
The syntax and associated semantics of the programming language used in the software development.

LANGUAGE IMPLEMENTATION
The mechanization of the language constructs in a machine representation which can be executed.

MODULARITY
The attribute of software that provides for the organization of the software into independent cooperating elements.

Table 3.3-3. Criteria for Software Quality Factors
(Page 3 of 3)

OPERABILITY
The attribute of software that determines the type and quantity of user procedures required to operate or interface with the software.

OPERATING SYSTEM ARCHITECTURE COMPATIBILITY
The degree to which operating system elements, their configuration, and their accessibility are effectively used by applications programs.

OPERATING SYSTEM INDEPENDENCE
The attribute of software which provides for the minimum direct interaction of developed software with specific operating system features.

SELF-DESCRIPTIVENESS
The attribute of software that provides for clarity and apparentness in describing the purpose or function of the software as well as the algorithm being used and its organization.

SIMPLICITY
The attribute of software which provides for the implementation in terms most easily understood.

TRACEABILITY
The attribute of software that provides for logical and structured connectivity from the highest level of specification to the source code implementation.

criteria are illustrated in Figure 3.3-1. These criteria are taken from /COOP79/ and minor additions have been made.

3.3.3 Software Quality Metrics

Software science is a term used by the late Maurice H. Halstead in /HALS77/ to describe a science that "deals only with those properties of algorithms that can be measured, either directly or indirectly, statically or dynamically, and with relationships among those properties that remain invariant under translation from one language to another." Although software science has also been applied to various textual materials, the application here will be to software developed using Ada.

3.3.3.1 Objectives

In any programming language, there are many different ways of representing an algorithm. Among those alternative representations some will be recognized as "poor," some as "good," some as "average," and some as "equivalent" by programmers fluent in the given language. The problem is that without quantitative measures, it is difficult to make meaningful comparisons based on common criteria.

The purpose of using selected software metrics as part of the Ada evaluation is to provide a common foundation for measuring certain properties of a given algorithm. It is anticipated that two circumstances will exist under which the software metrics will be used. First, in cases where notable difficulty was encountered in implementing particular algorithms (tasks, packages, subprograms in Ada) or portions thereof, alternative representations will be explored to determine if clearer, more compact or less difficult representations can be found. Second, as a result of software science efforts, several stylistic flaws, known as impurity classes, have been identified. By being able to identify them and relate them to the use or lack of use of Ada features, it will be possible to assess how Ada impacts on these flaws and what, if any, standards or guidelines are needed.

	SOFTWARE QUALITY FACTORS / SOFTWARE QUALITY CRITERIA	ACCURACY	COMMUNICATION COMMONALITY	COMMUNICATIVENESS	COMPLETENESS	CONCISENESS	CONSISTENCY	DATA COMMONALITY	ERROR MANAGEMENT	GENERALITY	HARDWARE ARCHITECTURE	HARDWARE INDEPENDENCE	INSTRUMENTATION	LANGUAGE CONSTRUCTS	LANGUAGE IMPLEMENTATION	MODULARITY	OPERABILITY	OPERATING SYSTEM ARCHITECTURE	OPERATING SYSTEM INDEPENDENCE	SELF-DESCRIPTIVENESS	SIMPLICITY	TRACEABILITY
SOFTWARE DEVELOPMENT	EFFICIENCY I					●								●								
	FLEXIBILITY									●		●				●				●		
	INTEROPERABILITY		●													●						
	MAINTAINABILITY					●	●	●								●		●		●	●	●
	REUSABILITY									●		●				●			●	●		
	TESTABILITY					●							●			●				●	●	
	TRANSPORTABILITY											●				●			●	●		
SOFTWARE PERFORMANCE	CORRECTNESS	●			●		●															●
	EFFICIENCY II										●				●			●				
	INTEGRITY	●				●											●			●	●	
	RELIABILITY	●	●	●			●	●	●								●				●	
	ROBUSTNESS						●		●	●												
	USABILITY			●													●					

SCI 2234U

Figure 3.3-1. Software Quality Factor-Criteria Interrelationships

3.3.3.2 Impurity Classes

Impurity classes are important to recognize for two reasons. First, to the extent that impurities remain in an algorithm, the software metrics calculated will be less reliable. Second, the existence of impurity errors in most cases appears to be an indication of less than "polished" code and thus an indication of potential lack of software quality in both the software development (e.g., maintainability, efficiency, testability) and the software performance (e.g., efficiency, robustness) areas. The definitions of six impurity classes which will be considered are given in Table 3.3-4.

3.3.3.3 Selected Software Metrics

In order to provide a quantitative evaluation of alternative Ada representations, several selected software metrics will be defined. The following definitions apply:

- η_1 - number of unique or distinct operators appearing in a specific implementation.
- η_2 - number of unique or distinct operands appearing in a specific implementation.
- N_1 - total usage of all of the operands appearing in that implementation.
- N_2 - total usage of all of the operands appearing in that implementation.

From these definitions, several metrics are defined and described below:

The implementation length, N, and estimated implementation length, $\hat{N}$, are given by

$$N = N_1 + N_2 \qquad (3.3\text{-}1)$$

$$\hat{N} = \eta_1 \log \eta_1 + \eta_2 \log \eta_2 \qquad (3.3\text{-}2)$$

in which log denotes the logarithm, base 2, unless otherwise noted.

Table 3.3-4 Impurity Classes

Type	Description	Example
I	Complementary Operations	Y = 2*T + X - T
II	Ambiguous Operands	X = PI*R**2 X = MI*Y + CONST
III	Synonomous Operand	T1 = P + Q T2 = P + Q R = T1 * T2
IV	Common Subexpressions	R = (P + Q) * (P + Q)
V	Unwarrented Assignment	T = P + Q R = T * T
VI	Unfactored Expressions	R = P*P + 2*P*Q + Q*Q vs. R = (P + Q)**2

The program or implementation size, called the volume, is given by

$$V = N\log\eta \qquad (3.3\text{-}3)$$

which gives the interpretation of volume in terms of bits with $\eta = \eta_1 + \eta_2$. An additional volume, known as the potential or minimal volume is given by

$$V^* = (2 + \eta_2^*)\log(2 + \eta_2^*) \qquad (3.3\text{-}4)$$

and denotes the most compact from (e.g., predefined subroutine) in which an algorithm could be represented with η_2^* representing the minimum number of unique input/output parameters.

The program level and estimated program level of an implementation are given as

$$L = V^*/V \ (L \leq 1) \qquad (3.3\text{-}5)$$

and

$$\hat{L} = (\eta_1^* / \eta_1)(\eta_2/N_2) \qquad (3.3\text{-}6)$$

$$= (2 / \eta_1)(\eta_2/N_2) \qquad (3.3\text{-}7)$$

provide a means of comparing alternative representations in the same implementation language where $\eta_1^* = 2$ by definition.

The "effort" required to implement a program, in terms of the total number of elementary discriminations, is given by

$$E = V/L = V^2/V^* \qquad (3.3\text{-}8)$$

and thus provides a means of measuring the effort required to implement the same algorithm in alternative ways.

Of the above equations, 3.3-4, 5, 6, and 8 will be used along with the identification and elimination of impurities to evaluate alternative Ada representations and determine the merits of the alternative representations.

3.3.4 Application-Oriented Requirements

The software quality parameters which were previously identified can be applied, in general, to any software irrespective of the application. But in addition to these parameters, there are also application-oriented requirements which a language must satisfy in order to

facilitate the development of software for the target applications. This section identifies the application-oriented requirements for the communication and trusted software requirements.

3.3.4.1 Communication Application Requirements

A previous study performed for the Defense Communication Agency /BBNI76/ resulted in the definition of the syntax and semantics of the Communications Oriented Language (COL). As part of that study, three alternative sets of requirements, which are desirable for a COL to have, were examined. The first set, obtained from the "U.S. Air Force HOL Standardization Study," is given in Table 3.3-5; the second set, obtained from "The Initial Report on the Suitability of JOVIAL for Communications Systems Implementation" is given in Table 3.3-6; the third set, obtained from "The Rome Air Development Center Report on Common-Communications Processors" is given in Table 3.3-7. As can readily be seen, there is commonality among the items of each set; however, there is also some discrepancy. Furthermore, each list is also a mixture of high-level language-inherent features as well as requirements for access to data, instructions, and controls at the machine level. From these lists a composite list of specific requirements shown in Table 3.3-8 was formed. This list will serve as a basis for assessing the efficiency and effectiveness of Ada as a language for developing communications software. The report also indicated some generalized requirements which are also shown in Table 3.3-8 .

3.3.4.2 Trusted Software Application Requirements

Unfortunately, it appears that no studies have been performed which identify explicitly a set of requirements that a language should possess for implementing trusted software. Upon examination of the application area, however, it is apparent that many desirable features are similar or identical to communication applications. Thus, to a large

Table 3.3-5. Communication Application Requirements (Set 1)*

a. Operating system functions
b. Access to timers
c. Bit manipulation
d. List processing
e. Character manipulation

* "U.S. Air Force HOL Standardization Study"

Table 3.3-6. Communication Application Requirements (Set 2)*

a. Capability to patch programs at the binary level in a rapid manner
b. Accessibility to the operating systems via privileged instructions
c. Run-time loading of a program from another program
d. Capability to shift to different random access devices
e. Capability to monitor the operation of individual programs in the system
f. Resolution of all relative addresses for overlay actions
g. Modification of various run-time parameters for assigning I/O devices
h. Handling of many and unique I/O devices
i. Processing of time critical events
j. Special requirements for automated recovery and accounting of messages

* "The Initial Report on the Suitability of JOVIAL for Communications System Implementation"

Table 3.3-7. Communication Application Requirements (Set 3)*

a. Modularity
b. Bit and byte access and manipulation
c. Interrupt-register access and manipulation
d. I/O device table generation
e. Real-time clock and interval timer access
f. A program-controlled interrupt capability
g. Communications channel control word access and manipulation
h. Insertion of machine language subroutines in the higher level language stream
i. Insertion of machine language instructions
j. Macro-generation
k. Diagnostic and debug statements

* "The Rome Air Development Center Report on Common-Communications Processors"

Table 3.3-8. Communication Application Requirements

General Requirements

a. Very high performance
b. Capability to interface with and manipulate specialized hardware
c. High portability of source code
d. Sophisticated data structures
e. Sophisticated control structures
f. Very high reliability

Specific Requirements

a. Bit and byte string access and manipulation
b. Insertion of assembly language code
c. Access to operating system functions and primatives
d. Access to and control of interrupts
e. Access to real-time clock and associated interval timer(s) or equivalent capability
f. Macro definition and generation
g. Access to debugging and diagnostic statements
h. Generation of I/O tables
i. Modularity
j. Parallel processing constructs
k. Strong data typing
l. Structured programming constructs

extent Table 3.3-9 is identical to Table 3.3-8 for communication applications.

In addition, however, two other features are believed to be strongly related to the characteristics inherent in trusted software. The first of these is data and control encapsulation. With this ability it should be possible to construct tamperproof data and control structures which can be used effectively but without knowledge of the details of the implementation and therefore, without the ability for unauthorized alteration or manipulation of the structures. The second is formal verification of the source code. Although this evaluation of Ada will not include formal verification of the trusted software source code, indications are that there is a strong correlation between the style in which programs are written and the ability to formally verify those programs /SRII78/. Also, there is a correlation between the style in which programs are written and the features provided by a language which encourages the writing of programs in a clear, intelligible, and verifiable style or at least proscribes certain undesirable styles.

3.3.5 Ada Language Features

In order to complete the evaluation of Ada with regard to producing qualify software, two additional areas of evaluation must be defined. The first of these is the use of the Ada features in a given application area; the second is the relationship between the Ada features and the software quality criteria previously defined.

Beginning with the second area, the Ada programming language includes many new language features which are available for the first time in a language designed for use on large-scale, embedded-computer-system, software projects. It is necessary to relate them to the software quality factors for two reasons. First, this will permit an assessment of Ada regarding which features affect which quality factors. Second, it will also provide an explicit identification of Ada features

Table 3.3-9. Trusted Software Application Requirements

General Requirements

a. Very high performance
b. Capability to interface with and manipulate specialized hardware
c. High portability of source code
d. Sophisticated data structures
e. sophisticated control structures
f. Very high reliability

Specific Requirements

a. Bit and byte string access and manipulation
b. Insertion of assembly language code
c. Access to operating system functions and primatives
d. Access to and control of interrupts
e. Access to real-time clock and associated interval timer(s) or equivalent capability
f. Macro definition and generation
g. Generation of I/O tables
h. Modularity
i. Parallel processing constructs
j. Strong data typing
k. Structured programming constructs
l. Data and control encapsulation (hiding)
m. Formal verification of source code

for reference in assessing the application software. The Ada language features are presented in Table 3.3-10 below and are divided into six categories which are data structures, data manipulation, modularity, concurrent programming, error management, and machine and implementation dependencies.

The next step will then be to evaluate the application software itself by determining the extent to which the Ada features have been used and the extent to which alternative features could have been used to achieve a better representation of the algorithm or data. In this context it will be important to identify which software factors are affected since, for example, maintainability may be improved at the expense of either development costs or execution efficiency.

3.4 SOFTWARE DEVELOPMENT STRUCTURE

3.4.1 General Approach

The general approach to the evaluation of Ada with regard to software development will consist of organizing a mini software development project for the SIP/ADCCP and ACCAT GUARD applications, collecting data related to the software quality factors on each application as it progresses through the various software development phases and providing a nominal set of software development standards and guidelines which are consistent with MIL-STD-1679 (NAVY) /M16778/.

The intent of this approach is to have the software developed under circumstances which, as nearly as possible and practicable, duplicate those of a major software development project. The reasons for this are twofold: first, to provide a comprehensive Ada evaluation, it is desirable to utilize as many Ada features as possible; second, in order for the results to have validity when extrapolated to large software development projects, this effort should be as representative in kind as possible.

Table 3.3-10. Ada Language Features (Page 1 of 3)

Data Declaration
- Data Abstraction
  - Type declarations
  - Subtype declarations
  - Overloading
  - Aliasing
  - Attributes
  - Renaming of objects
- Data Checking
  - Strong typing
  - Mode declaration for formal parameters

Data Manipulation
- Aggregrates
  - Arrays
  - Records
  - Variant Records
- Unchecked Programming
  - Object deallocation
  - Type conversions
- Overloading
  - Subprograms
  - Operators
- Structured programming constructs
- Attributes
- Dot notation for object referencing
- Dot notation for component referencing in records
- Index notation for component referencing in arrays
- Object creation via allocators

Table 3.3-10. Ada Language Features (Page 2 of 3)

- Modularity
    - Modules
        - Program units
        - Compilation/Library units
        - Compilation Subunits
        - Generic unit definition
        - Generic unit instantiation
        - Separation of Specifications and bodies
    - Encapsulation of data/controls
    - Importing of modules
    - Blocks

- Concurrent Programming
    - Task definition
    - Task interaction control
        - Rendezvous
        - Selective wait
        - Conditional entry call
        - Timed entry call
    - Task attribute definitions
    - Task activation/termination
    - Task priorities

- Visibility Control
    - Scope declarations
    - Renaming declarations
    - Direct visibility
    - Qualified visibility
    - Private types
    - Limited private types

Table 3.3-10. Ada Language Features (Page 3 of 3)

- Error Management
    - Internally defined error conditions
    - Exception processing
        - Declaration
        - Raising
        - Handling
        - Propagation

- Machine and Implementation Dependencies
    - Pragmas
    - STANDARD package
    - SYSTEM package
    - Data representation control
        - Length specifications
        - Enumeration type representations
        - Record type representations
        - Multiple representations
    - Address/interrupt control
    - Machine code insertion
    - Foreign code interface
    - Input/output

3.4.2 Design Phase

The design phase of Phase II will use the stepwise-refinement design approach consisting of two design steps using Ada as the design language. The first step or portion will consist of establishing the macroscopic software designs; the second step will consist of refining those designs sufficiently to permit completion of the code.

The macroscopic design portion will focus on using existing Program Performance Specifications or Computer Program Development Specifications (Type B-5) as the basis for designing the SIP/ADCCP and ACCAT GUARD applications. This information will be supplemented with additional or changed requirements in the case of ACCAT GUARD to account for the facts that the original implementation was on a Western Electric UNIX-based system and that the design was modified. The objective of the macroscopic design phase will be to establish all program modules (packages, tasks, subprograms, compilation units and subunits, and their dependencies), the definition of all formal parameters used as module inputs or outputs, and the definition of abstract data types for inputs, outputs, and global and common data. In some instances, major decisions within a module may also be indicated as a means of delineating overall control flow. Finally, lists of called and calling modules will be formed for each module. In accomplishing the macroscopic design, a proper subset of Ada constructs will be used as a design or specification language and will result in modules which can be compiled and error-checked. The objectives here are to gain an early, increased understanding of Ada without the need to consider irrelevant details and to, as early as possible, orient the designs of the applications to the Ada language features.

The microscopic design portion will modify the macroscopic designs as required and refine them to the next level of detail. This level of detail will include the definition of the components of the abstract data types, the refinement of all global or common data objects (as opposed to strictly local) including preset values, and the specification

of all major control decisions within each module. As during the macroscopic designs, the refined modules will be compiled to achieve error-checking of the refined design.

During the microscopic design phase composite test plans/procedures will be produced which will define the tests to be performed in debugging and integrating the software.

The detailed design phase will be conducted in accordance with the software development standards identified below. The design phase will include an informal Preliminary Design Review (PDR) and a Critical Design Review (CDR) with the objective of highlighting any difficulties encountered during the design.

3.4.3 Code/Debug/Modify Phase

The code and debug phase will consist of translating the microscopic designs into Ada code, compiling the code and removing compilation errors, desk-checking the code, and performing the tests defined in the test plans/procedures.

During the modify portion of this phase, the programming of one or more application modules will be shifted to the person responsible for the other application. The objective of this shift is to duplicate the circumstances surrounding software maintenance in which the maintenance personnel had no previous involvement with the project. This will also provide preliminary familiarization with the other application and give the basis for subsequent participation in the software evaluation.

3.4.4 Integration and Test Phase

The integration and test phase will consist of producing the required, completed program for each application. This will include conducting program and, if required, system integration tests according to the test plans/procedures, and integrating all software elements into a complete program which is ready for performance evaluation.

3.4.5 Test Software Development

Specific test support software which needs to be developed for the SIP/ADCCP and ACCAT GUARD applications has been identified in Section 7.3. The software will be designed using the macroscopic/microscopic approach established for the application software and will be coded during the code/debug/modify portion of the software development.

3.4.6 Software Development Standards

The software development guidelines of Sections 5.3, 5.4, 5.5, 5.6, and 5.8 of MIL-STD-1679 (NAVY) /M16778/ will be used in a nominal manner consistent with the software development effort and incorporated as part of the Software Development/Management Plan.

3.5 DATA ACQUISITION AND ANALYSIS

The data acquisition and analysis portion will be concerned with obtaining data from three sources for use in the analysis and evaluation of Ada. These souces are error statistics, the structure of the developed software, and programmer interviews.

3.5.1 Error Statistics

The error statistics to be compiled comprise two groups which are compilation-related errors and execution-related errors. The objectives in collecting these error statistics are: 1) to determine if there are any particular Ada constructs or sequences of constructs which seem to be systematically difficult to use, 2) to determine which type(s) of errors, if any, remain hidden following a successful compilation and must be detected during execution, 3) to relate errors to module complexity, and 4) to help in the identification of guidelines and alternatives which will either diminish or remove the problem-causing areas which are deemed most severe.

For compilation-related errors, the errors encountered for each compilation unit or subunit will be identified by type and frequency of occurrence. Additionally, the total number of compilations per compilation unit or subunit will also be maintained.

For execution-related errors, the errors detected via unanticipated exceptions and erroneous (inaccurate, incomplete, inconsistent) computational results will be similarly grouped by type and frequency of occurrence.

3.5.2 Software Structure

The primary objective of the software structure analysis is to determine which Ada features were used and to assess the degree of success or difficulty encountered in their use. The secondary objective is to assess in a qualitative and, if possible, quantitative manner the effectiveness and suitability of the features used.

To accomplish the first objective, the software will be examined at two levels. The first level will address the overall organization of the software into modules comprising packages, subprograms, tasks and compilation units and subunits. This organization will be compared with the totality of Ada features and with the software quality factors in order to determine how "good" or suitable the structure is. The second level will address the internal organization of the data structures and bodies of the various modules for the purpose of assessing the breadth of the Ada features used as well as determining the overall composition of the features used. Of particular concern here will be whether full advantage was taken of the Ada features or whether a subset of Ada features was used in the style of some other language. To accomplish the second objective, the Ada features used within each module will be analyzed. In those cases where a particular Ada feature, construct, or set of constructs appears to be awkward or suboptimal regarding efficient representation in Ada, or especially difficiult to implement or understand, a detailed review of the constructs will be made with a view

toward finding alternate, improved representations. For those cases in which alternative representations are found, the software metrics previously defined will be used to evaluate some of the merits of each alternative.

3.5.3 Programmer Interviews

The third source of data will be interviews conducted with the programmers who implemented the SIP/ADCCP and ACCAT GUARD software. The overall objective of these interviews will be to elicit qualitative information regarding Ada. First, information will be obtained regarding both the suitability of the Ada features with respect to the type of applications implemented and the limitations and unwise use of Ada features. Second, a cross-perspective of two potentially different design and implementation approaches will be obtained by having one programmer implement a small portion of the other's design as a means of assessing maintainability issues. Third, an attempt will be made to understand the rationale applied in the design and development phase for those approaches which worked, as well as those approaches which had problems. An additional result of this understanding should be the ability to formulate new and improved approaches to design and implementation using Ada.

3.5.4 Data Acquisition and Analysis Procedures

The sections above have identified the three sources from which data will be extracted and analyzed. During the early portions of Phase II, the detailed data acquisition and analysis procedures will be formed. For the error statistics, the Ada/ED compilation errors will be identified and divided into various classes so that error types and frequencies of each module can be readily identified and associated with that module. A similar classification of run-time errors will be established. In conjunction with the error statistics, the software modules will also be ordered by complexity ranging from arithmetic computations (least complex) to input/output and operating system functions (most complex).

In the software structure area, the software metrics identified previously will be compared against the Ada language constructs to establish consistent and unambiguous procedures for counting the program operators and operands in those cases in which detailed quantitative analysis of the software structure will be performed. In addition, the Ada language features such as data abstraction and overloading will be related to the software quality criteria so as to identify explicitly relationships and trade-offs between the features and the various software quality criteria.

For the programmer interviews, questionnaires will be formed to obtain qualitative assessments of the various Ada features. Procedures or methods will then be established which relate those assessments to the established software quality factors and criteria.

3.6 SOFTWARE TESTS

As indicated above, two levels of testing will be performed. These comprise module testing and system integration testing. The objective of the module testing is to exercise each module so as to assure that all internal program errors have been detected and corrected prior to system integration testing. The objective of the system integration testing is to combine all software for each application, including the test support software, and exercise the software through the use of the functionally oriented system integration tests. The functional tests for each of the software applications are defined below.

3.6.1 SIP/ADCCP Software Tests

The SIP/ADCCP system integration tests will comprise three groups of tests which are the SIP, ADCCP, and line control module (LCM) tests. The SIP tests will include the simulation of missing segments, duplicate segments, and segment checksum errors. The ADCCP tests will include the simulation of out-of-sequence packets, controls, commands and responses, time-outs, and invalid-frame errors. The line

control module tests will include the insertion of time-out errors, CRC errors, and data errors as the data is transferred on an inter-ADCCP mode.

3.6.2 ACCAT GUARD Software Tests

The ACCAT GUARD System integration tests will be designed to exercise the ACCAT GUARD functional capabilities. The specific tests to be conducted include high-low and low-high mail transfers, the use of the free-style English language queries in the high-low query and response and low-high query and response transfers. (Canonical queries (preformatted data base queries) will not be used because there is no actual high-host or low-host data base and because they are, in effect, a subset of the free-style English language queries.) Two additional functional tests will include the review of information for downgrading (accept or reject) by the Security Watch Officer (SWO) and the sanitization of high-low transfers by the Sanitization Personnel (SP) for downgrading requests rejected by the SWO.

In addition to the execution testing of the ACCAT GUARD software, the source code of the Upgrade Trusted Process (UGTP) and the Downgrade Trusted Process (DGTP) will undergo an implementation correspondence test with the formal specification of the trusted processes. This will be done as an additional means of both detecting errors and verifying general correspondence between the implementation and the formal specifications for the trusted processes. To the extent possible, an attempt will also be made to assess the viability of formally verifying the code as would be done if an Ada verifier were available.

3.6.3 Software Performance Tests

As indicated earlier, the use of the Ada/ED language translator will preclude the evaluation of certain software performance quality factors. Software performance quality factors which can be evaluated within the capabilities of Ada/ED are correctness, integrity, reliability, and robustness.

The Efficiency II factor, memory, and execution efficiency can be evaluated only with the use of a native code compiler which may include the capabilities for selected memory space or execution speed optimization. Examples of tests which should be conducted are event timing (accuracy and repeatability), error management alternatives (error propagation vs. handling at source), consequences of system-initiated vs. user-controlled garbage collection, consequences of task creation via task types vs. use of anonymous tasks, effects of optimized vs. non-optimized code, task interaction delays using the various tasking constructs, effects of priority on processing and rendezvous, and memory utilization of alternative data structures.

3.7 ADA EVALUATION RESULTS

At the conclusion of the project the results and findings of the project will be documented in the Development/Performance Evaluation report in two categories; the project summary and the programming standards and guidelines.

3.7.1 Project Summary

The project summary will provide several types of information regarding the project as a whole. This information will include an assessment of the suitability of the software development structure followed throughout Phase II, an identification of impacts caused by the immaturity of some software tools, the lack of an Ada Programming Support Environment, the results of implementing only portions of the ACCAT GUARD application, and similar project-related assessments. The objective here is to identify and separate those aspects of the project, if any, which may impact on the final results, but are not inherent in the Ada language itself. Secondary objectives are to document the progress of the project as a means of identifying which alternatives were selected, why, and what their consequences were in terms of the project structure and identifying Ada language problems (syntax

or semantics) which should be reviewed with regard to modification.

3.7.2 Ada Programming Standards and Guidelines

As stated previously, the primary objective of Phase II, the test and evaluation phase, is to evaluate the suitability of Ada for producing communications and trusted software. Because of the many new features provided in Ada, it will be possible to produce software with a new degree of sophistication and complexity. Conversely, with the sophistication of the Ada constructs, it is also necessary to assure that the constructs are used in a controlled manner so that the overall software quality objectives will be achieved. Thus, as a result of the data acquisition and data analysis performed during the latter portion of Phase II, a set of programming standards and guidelines will be formed. These standards and guidelines will be designed to specifically indicate what control and usage measures should be implemented over and above the capabilities provided by the Ada language to assure the consistent, effective and efficient use of Ada in a production, software-development environment.

(This Page Intentionally Left Blank)

SECTION 4
TEST AND EVALUATION MANAGEMENT REQUIREMENTS

4.1 CONTRACTOR RESPONSIBILITIES

The contractor responsibilities defined below are identical to those tasks specified in the schedule in Section 8. All documentation except the final version of the Development/Performance Evaluation Report will be produced as draft reports for review by the COR.

4.1.1 Software Development/Management Plan

SCI shall produce a draft Software Development/Management Plan. The purpose of this plan is to establish preliminary programming standards which are consistent with MIL-STD-1679 /M16778/ and to completely specify the extent of the software to be developed, to identify the environment in which the software will be developed and to establish specific procedures which will be used to monitor and control the software development.

As an adjunct to the Software Development/Management Plan, SCI will conduct an Ada language indoctrination. The purposes of this indoctrination will be to emphasize maximal use of the Ada features, emphasize the objectives of the macroscopic and microscopic design efforts, establish guidelines with respect to any compiler limitations which may exist, and to formally review and evaluate existing documents and research results which may have a bearing on the "best" use of Ada.

4.1.2 Ada/ED Delivery/Installation

At the beginning of Phase II SCI shall initiate the request for the Ada/ED translator-interpreter and corresponding documentation and shall supply the necessary magnetic tape(s) for obtaining Ada/ED and all required support software.

Following delivery of Ada/ED and the documentation, SCI shall host the Ada/ED on the VAX 11/780 located at the University of California San Diego (UCSD) Computer Center (CC) and verify that Ada/ED is functioning properly. SCI shall also initiate the necessary contracting/purchasing procedures with the UCSD CC for the use of the VAX 11/780 facility and associated services.

4.1.3 Software Design

During the software design stage SCI shall perform the macroscopic and microscopic designs for the SIP/ADCCP and ACCAT GUARD applications and shall produce drafts of test plans/procedures to be used in testing the software.

4.1.3.1 Macroscopic Software Design

In the macroscopic software design stage, SCI shall translate the software requirements for the SIP/ADCCP and ACCAT GUARD applications into macroscopic (high-level) designs. This shall be accomplished by using a proper subset of the Ada constructs to represent the macroscopic designs.

4.1.3.2 Microscopic Software Design

In the microscopic software design stage SCI shall translate the macroscopic software designs into a sufficient level of detail such that completed code can be produced during the Code/Debug/Modify stage.

4.1.3.3 Test Plans/Procedures

During this stage SCI shall produce draft versions of a test plan/procedures for the SIP/ADCCP and ACCAT GUARD applications. These plans/procedures shall identify test software to be used and test cases to be performed to determine the correctness of the developed Ada programs with emphasis on the system integration testing.

4.1.3.4 Design Review

SCI shall conduct two design reviews. The first design review shall be conducted at the conclusion of the macroscopic design to assure that all functional capabilities have been addressed. The second design review shall be conducted at the conclusion of the microscopic design to assure that the detailed design provides for the best use of the Ada features.

4.1.4 Software Development

During the code/debug portion SCI shall translate the microscopic designs for SIP/ADCCP and ACCAT GUARD into Ada code which can be compiled and debugged. Similarly, test drivers which are needed to exercise the applications shall also be coded and debugged. As significant portions of the code become available SCI shall integrate and test them.

During the modify portion of this stage, modifications or additions will be made to existing code to assess the maintainability of the code.

4.1.5 Evaluation Procedures

During this stage, SCI shall produce the detailed software-development and software-performance evaluation procedures based upon the requirements of Section 3. Because of the limitations of the planned Ada compiler, the emphasis shall be placed on the software-development instead of the software-performance evaluation.

4.1.5.1 Software-Development Evaluation Procedures

SCI shall translate the software-development evaluation requirements into specific procedures and data formats which will readily permit the necessary data to be obtained during the software development effort.

4.1.5.2 Software-Performance Evaluation Procedures

SCI shall translate the software-performance evaluation requirements into specific procedures and data formats which will readily permit the necessary data to be obtained during the software-performance (testing and integration) effort.

4.1.6 Data Acquisition

SCI shall extract the data required to assess the effectiveness of Ada as a programming language for communications and trusted software applications. The data will be acquired from three sources which are error statistics, software statistics and analysis, and programmer interviews.

4.1.7 Data Analysis

SCI shall perform an analysis of the extracted data in accordance with the evaluation requirements and procedures. This analysis shall be designed to identify any efficiency or effectiveness criteria regarding the use of Ada for communications or trusted software applications. In addition, any specific Ada-related problems shall also be identified and a set of guidelines or standards shall be provided which indicate the best use of Ada in the two application areas.

4.1.8 Development/Performance Evaluation Report

SCI shall present summaries of the data collected and results of the data analysis through a Development/Performance Evaluation Report. The preliminary data and results will be compiled into a draft report which shall be submitted to the Contract Officer's Representative (COR) for review, comment and approval. SCI shall then produce a final report which incorporates any corrections, additions or changes. The report shall be produced according to MIL-STD-847A, 31 January 1973, "Format Requirements for Scientific and Technical Reports Prepared by and for the Department of Defense" /M84773/.

4.1.9 Software Delivery

At the conclusion of Phase II, SCI shall rehost the source and executable test software and any other support software which was developed to the VAX 11/780 located at the Defense Communications Engineering Center in Reston, Virginia. A summary users guide will be provided with the delivered software and final drafts of produced documents will be delivered also.

4.2 PROCURING AGENCY RESPONSIBILITIES

4.2.1 Software Development/Management Plan Review

As part of the evaluation effort an abbreviated Software Development/Management Plan will be produced. This plan will be submitted to the COR for review, comment, and approval early in the design portion of the evaluation.

4.2.2 Software Design Review

The macroscopic and microscopic software designs will be submitted to the COR for review and comment prior to the conduct of the planned SCI design review. In addition, the COR will be invited to participate in the actual design review process if he so chooses.

4.2.3 Test Plans/Procedures Review

As part of the design and development portion of the evaluation, combined test plans/procedures will be produced to test the developed software with regard to correctness. These plans will be submitted to the COR for review, comment, and approval prior to the initiation of the integration and test effort.

4.2.4 Development/Performance Evaluation Plans/Procedures Review

If during the course of the Ada evaluation problems occur which necessitate a change in the evaluation plans/procedures, the changes will be documented and submitted to the COR for review, comment and approval prior to proceeding with them.

4.2.5 Development/Performance Evaluation Report Review

After the development and performance evaluation data have been collected and analyzed, a draft of the Development/Performance Evaluation Report will be produced and submitted to the COR for review and comment. Comments and suggestions will be incorporated into the final report which will be delivered at the end of the contract.

4.2.6 Ada Language Processor

The U.S. Army Communications Research and Development Command (CORADCOM) CENTACS is currently sponsoring the development of an Ada "compiler" by the Courant Institute of Mathematical Science (CIMS) of New York University. It is recommended that this "compiler" be obtained by the Defense Communication Agency for use under this Evaluation Plan.

The Ada language processor being developed has been designated Ada/ED and is mechanized as a translator-interpreter which has been coded in SETL and is hosted and targeted on a Digital Equipment Corporation (DEC) VAX 11/780.

Ada/ED is planned for public release in April 1981. A subsequent, planned release will be directed at improving the throughput of Ada/ED. Because of the translator-interpreter mechanization of Ada/ED it will not produce native code for the VAX 11/780. Thus, it will not be possible to obtain or project software-performance statistics relating to optimizing, production-quality compilers which are being designed and built to produce native code.

The U.S. Army will provide user documentation on the hosting of Ada/ED and its operation. This documentation will be required for the development of the software identified in this Evaluation Plan. Also, in order to minimize impact of problems on Evaluation Plan efforts, a mechanism will be established to remain informed of Ada/ED problems and planned new versions or releases.

The Ada/ED translator-interpreter will be provided as a complete software package which includes all supporting software written in SETL. The software will be supplied on 9-track/1600BPI magnetic tape which is suitable for hosting and execution of VAX 11/780 operating under VMS 2.1.

4.3 OTHER AGENCY RESPONSIBILITIES

No other agencies will be required in support of this effort.

4.4 ASSOCIATED SUPPLIER RESPONSIBILITIES

No associated suppliers will be required in support of this effort.

(This Page Intentionally Left Blank)

SECTION 5
PERSONNEL REQUIREMENTS

5.1 PROJECT MANAGEMENT PERSONNEL

A senior program engineer will be required as a Project Manager, to provide customer liaison, to coordinate all project activities which interface with other agencies or organizations, to direct the software design, coding, debugging and evaluation, and to report on project technical and financial status.

In addition to the management responsibilities, the senior program engineer will be responsible for producing the Software Development/Management Plan, defining the data acquisition procedures, collecting the data for subsequent analysis, analyzing the data and produced software, and contributing to the writing of the Development/Performance Evaluation Report.

5.2 SOFTWARE DEVELOPMENT AND EVALUATION PERSONNEL

Two senior system analysts will be required to design, develop, and evaluate the Ada software.

5.2.1 SIP/ADCCP Software Personnel

The senior system analyst assigned to the SIP/ADCCP software development effort will be responsible for producing the macroscopic and microscopic software designs, developing and debugging the code, developing the associated test plans and procedures, and conducting the software tests. He will also be responsible for integrating all software so that a comprehensive evaluation of the SIP/ADCCP software can be conducted with respect to the development and performance criteria.

This senior system analyst will also participate in the evaluation of the SIP/ADCCP and ACCAT GUARD software and will be assigned to code/debug a portion of the ACCAT GUARD software from the established design in order to help assess

certain software-development quality factors such as maintainability.

5.2.2 ACCAT GUARD Software Personnel

The senior system analyst assigned to the ACCAT GUARD software development effort will be responsible for producing the macroscopic and microscopic software designs, developing and debugging the code, developing the associated test plans and procedures, and conducting the software tests. He will also be responsible for integrating all software so that a comprehensive evaluation of the ACCAT GUARD software can be conducted with respect to the development and performance criteria.

This senior system analyst will also participate in the evaluation of the SIP/ADCCP and ACCAT GUARD software and will be assigned to code/debug a portion of the SIP/ADCCP software from the established design in order to help assess certain software development quality factors such as maintainability.

SECTION 6
HARDWARE REQUIREMENTS

6.1 DEVELOPMENT SYSTEM

The software will be developed on a VAX 11/780 located at the Computer Center (CC) of the University of California, San Diego. This facility has dialup, remote access and is within 15 miles of SCI's facilities.

The CC operates a VAX 11/780 with 2.25M bytes of memory under the VMS 2.1 operating system.

The VAX 11/780 is supported by 9-track 800/1600 BPI tape drives, REPO6-AA disks and has 24 dialup ports which can be operated at either 300 or 1200 baud.

The VAX 11/780 is fully supported by an operator for tape and printer services from 0800 to 0100, Monday through Friday and operates in an unattended mode during other times. A full range of user services is also provided including analysis and programming support, data preparation, dispatchers, hardware maintenance personnel and system support personnel. Several terminals are available at SCI's facility. These include LSI's ADM Information Display, Teletype Model 43, TI Silent 700 and IBM 3101.

6.2 DEMONSTRATION SYSTEM

The VAX 11/780 located at the Defense Communications Engineering Center in Reston, Virginia will serve as the demonstration system to demonstrate the developed software for the COR. This system will also serve as the system for rehosting the software at the conclusion of the contract.

(This Page Intentionally Left Blank)

SECTION 7
SUPPORTING SOFTWARE REQUIREMENTS

7.1 SYSTEMS SOFTWARE

The following VAX system software, which is supported by DEC, may be used either during the software development effort or during the software test/evaluation effort:

- VMS 2.1 - Operating System
- SOS - Interactive Text Editor
- SCP - Batch (Programmed) Text Editor
- MACRO - Macro-assembler
- LINKER - Object module linker
- LIBRARIAN - Object module librarian
- SORT - Native-code sort utility
- LIBRARY - Common run-time library

7.2 ADA PROGRAMMING SUPPORT SOFTWARE

The Ada/ED software will consist of the Ada translator-interpreter and supporting SETL routines. This software will be provided to the Defense Communications Agency in object format on magnetic tape by the U.S. Army, CORADCOM. The Ada/ED translator-interpreter will operate as an application program on the VAX.

7.3 TEST SUPPORT SOFTWARE

After the software has progressed to the integration and test portion of the development, certain additional software will be required to simulate inputs to and collect outputs from the software undergoing evaluation.

Software areas which will require this support are identified below. Specific software requirements will be identified during the software design effort and implemented as part of the debug, test and integration effort.

7.3.1 SIP/ADCCP Test Support Software

The SIP and ADCCP functions in the AUTODIN II configuration are shown in Figure 3.2-1. Since the software development effort entails only the SIP and ADCCP software, those functions will have to be supported with test support software. Figure 7.3-1 indicates the SIP/ADCCP test and evaluation configuration. The test support software, which will be developed to exercise the SIP/ADCCP software as integral components, is indicated by asterisks "*". This test support software consists of two components which are the Terminal Subscriber Interface and the Pseudo Line Control Module.

7.3.1.1 Terminal Subscriber Interface

The Terminal Subscriber Interface (TSI) will provide the interface between a "network" user accessing the "network" from a CRT-type device and the SIP/ADCCP software. In this test configuration the user will act as both the source and destination of messages.

Two specific functions will be performed by the TSI. First, the TSI will provide the user with the capability for entering and examining messages as well as controlling the transmission and receipt of messages. Such functions will consist of sending messages of various sizes, sending single or multiple messages and controlling the receipt of messages. Second, the TSI will provide the user with the capability to introduce various types of errors into transmitted packets via the Pseudo Line Control Module which is described below.

7.3.1.2 Pseudo Line Control Module

The Pseudo Line Control Module (PLCM) software will serve two purposes. First, the PLCM will act as a pseudo-network which will permit messages to be sent and received through the "networks" thereby being able to exercise the SIP/ADCCP in the full duplex transmission mode. Second, the PLCM will also be used to introduce various types of errors into packets which are transiting the "network". This will

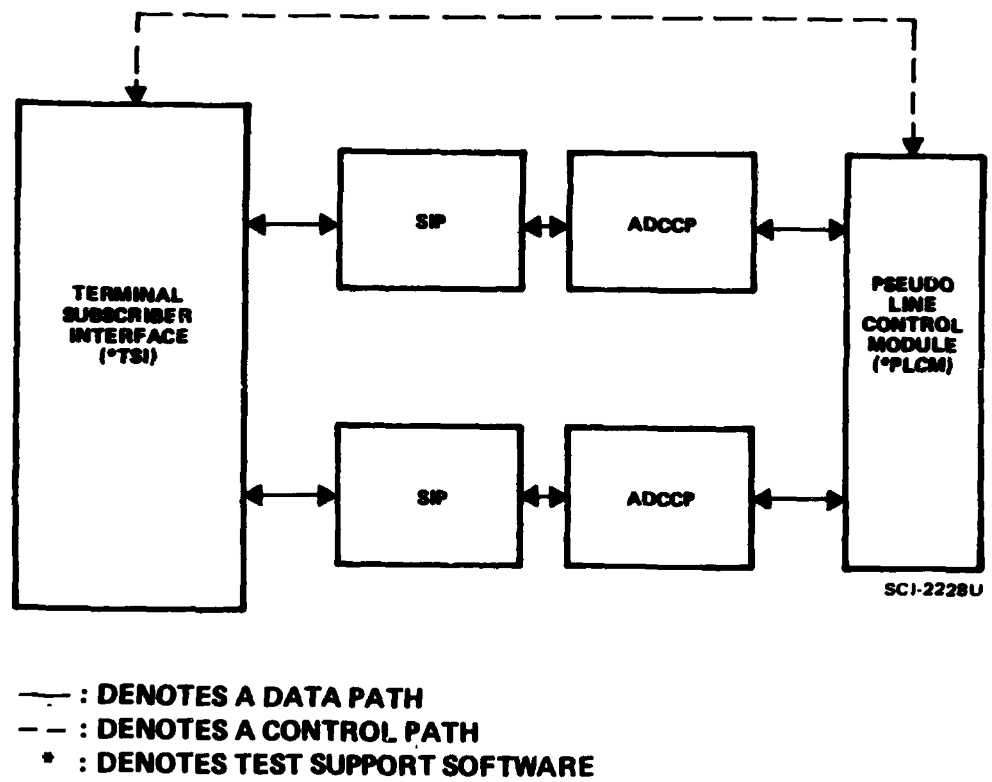


Figure 7.3-1. SIP/ADCCP Test and Evaluation Configuration

enable the SIP/ADCCP software to undergo moderately extensive testing with regard to the software performance factors of correctness, reliability, robustness, and integrity.

The error injection process at the PLCM will be under the control of the TSI software with different error conditions to be selected by the user. Such errors will include CRC errors, data errors, invalid frame errors, time-out errors, out-of-sequence packets and out-of-context responses and commands.

7.3.2 ACCAT GUARD Test Support Software

The ACCAT GUARD configuration showing the ACCAT GUARD system and the interfaces to the high-level and low-level networks is shown in Figure 3.2-2. The ACCAT GUARD software in its present configuration is shown in Figure 3.2-3. Of the thirteen distinct processes and one aggregate process (HGO), only two processes (DGTP and UGTP) comprise trusted software. However, in order to simplify interfacing and provide a more comprehensive and realistic software development evaluation, four other processes (HGSD, HDGD, LGSD, and SPCI) will also be implemented. The other interfaces with these processes will be implemented with test support software indicated by an asterisk "*", as shown in Figure 7.3-2. The functions of the GUARD test support software are described below.

7.3.2.1 High-Level Input/Output

The High-Level Input/Output (HLIO) module will be used to simulate the interface between ACCAT GUARD and the high-level network. Thus, this module will replace the functional operations relating to intersystem data flow which are performed by the existing HFS, HDP, HMD, and HDMD processes.

The high-level network will be represented through a combination of a high-level user terminal interface and files which are used to generate and release external high-level data and to receive and examine received data. Externally supplied data (inputs) will consist of mail and queries to be transmitted to the low-level network. Internally

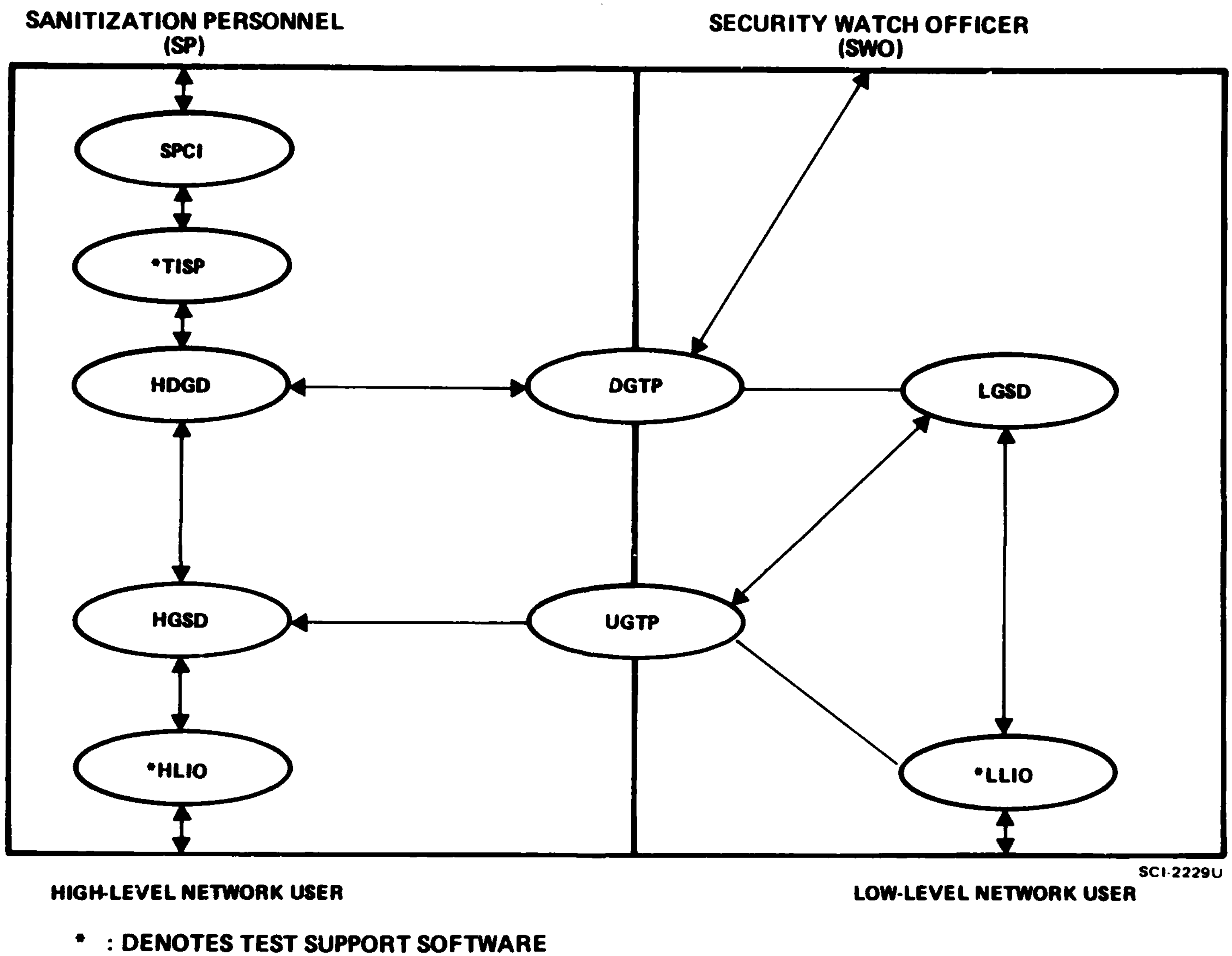


Figure 7.3-2. ACCAT GUARD Test and Evaluation Configuration

supplied data (outputs) will consist of mail and query responses received from the low-level network. In addition to the data exchanged, various control and status information will also be exchanged.

7.3.2.2 <u>Low-Level Input/Output</u>

The Low-Level Input/Output module will be used to simulate the interface between ACCAT GUARD and the low-level network. Thus, this module will replace the functional operations relating to intersystem data flow which are performed by the existing LFS, LDP, LMD, and LDMD processes.

The low-level network will be represented through a combination of a low-level user terminal interface and files which are used to generate and release external, low-level data and to receive and examine received high-level data. Externally supplied data (inputs) will consist of mail and queries to be transmitted to a high-level network; internally supplied data (outputs) will consist of mail and query responses received from the high-level network. In addition to the data exchanged, various control and status information will also be exchanged.

7.3.2.3 <u>Terminal Interface/Sanitization Personnel</u>

The Terminal Interface/Sanitization Personnel (TI/SP) module will be used to perform the functions of the processes identified within the HGO module and will interface with the SPCI and HDGD modules. Of the functions identified within the HGO module, only the following functions will be implemented totally or in part via the TI/SP module to allow the SP to function in a quasi-realistic manner: CONTROL, DELETE, EDIT, LIST, LOGOUT, LOGIN, NEXT, RELEASE, and SANITIZE.

SECTION 8
SCHEDULE

8.1 ADA EVALUATION SCHEDULE

The development schedule showing the planned Phase II tasks is given below in Figure 8-1.

Evaluation of Efficiency II, the memory and execution performance factor, has been explicitly indicated on the schedule since there is no plan to perform such an evaluation at this time due to the lack of an Ada compiler which generates native code.

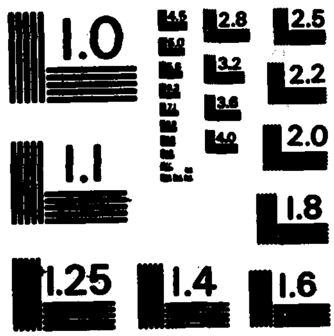
AD-A121 938 EVALUATION OF ADA AS A COMMUNICATIONS PROGRAMMING LANGUAGE(U) SYSCON CORP SAN DIEGO CA A L BRINTZENHOFF ET AL. 31 MAR 81 DCA100-80-C-0037

UNCLASSIFIED F/G 9/2 NL

END

FILMED

DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

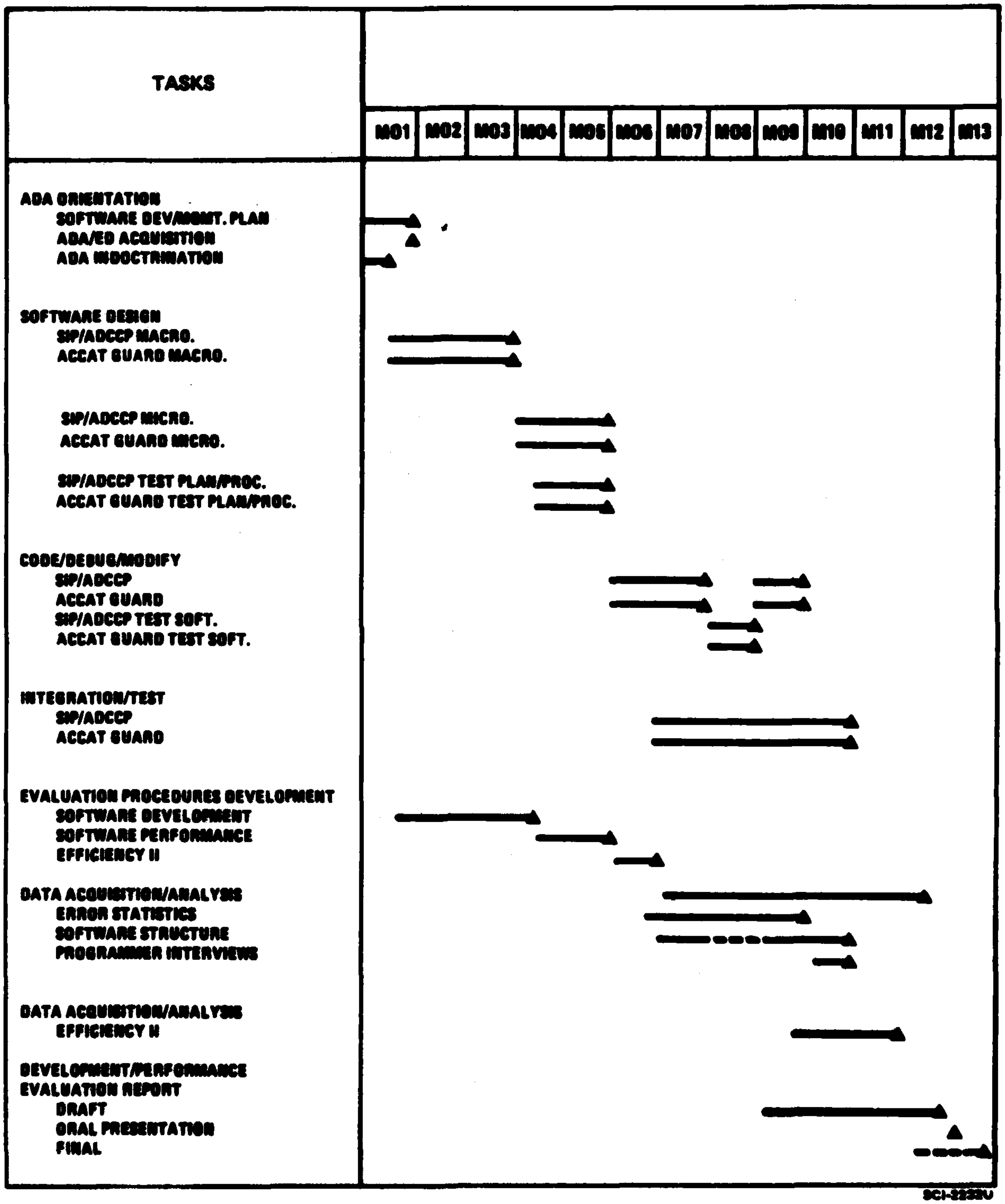


Figure 8-1. Ada Evaluation Schedule

SECTION 9
QUALITY ASSURANCE

9.1 QUALITY ASSURANCE OBJECTIVES

Typically, the quality assurance objectives or requirements portion of a test plan is to define the necessary testing controls, configuration management procedures, pass/fail criteria and overall management-review procedures relating to the conduct of the testing. In the context of the Phase II objectives of evaluating the suitability of Ada for developing communications and trusted software, the objectives will be shifted somewhat. First, rigid configuration management procedures will not be established since the objective is not to retain rigid control of production software and documentation. Second, pass/fail criteria will be established during the macroscopic design portion of Phase II because detailed modifications may need to be made to the SIP/ADCCP software and modifications will need to be made to the ACCAT GUARD software to make the evaluation effort tractable. Third, all compilations, design notes, test results and other project documentation will be retained throughout the entire project in order to provide a record of what decisions were made and why. Fourth, since the objective of the evaluation is to formulate standards and guidelines for using Ada, only minimal, initial programming guidelines and standards will be formed. These will be used primarily to focus on the Ada features as they relate to the applications with emphasis placed on using Ada as a new tool and not an old tool. This approach will also allow the maximum opportunity for innovative use of the Ada features. Finally, since this is an evaluation of Ada and not the developed software per se, the quality assurance emphasis will be placed on maintaining historical data as the development progresses and on providing interim reviews both for the purpose of measuring progress and for providing early and continuing opportunities for customer review.

9.2 QUALITY ASSURANCE REVIEWS

Several intermediate milestones will be planned to provide interim reviews of progress and results. First, since it is planned that Ada will be used as the design language as well as the implementation language the first significant review will occur at the conclusion of the software design phase. The second significant review will follow the interviews with the programmers who developed the SIP/ADCCP and ACCAT GUARD software. The third review will coincide with the completion of the draft of the final report and will provide a significant review opportunity for the DCA prior to the oral presentation. The fourth review will be in the form of an oral presentation which will provide an opportunity for discussion of the draft report, present any additional information, and provide for interactive discussion of the preliminary results.